ANALYZING COMMUNICATION FLOW AND PROCESS PLACEMENT IN LINDA PROGRAMS ON TRANSPUTERS

THESIS

Submitted in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF SCIENCE (APPLIED COMPUTER SCIENCE)
OF Rhodes University

by

Frederick Kofi de-Heer-Menlah

June 1991

To my mum Emelia

Acknowledgements

Thanks to my supervisors, Dr Peter E. Wentworth and Professor Peter Clayton for their guidance, and the ideas they wilfully imparted during this research. My sincere thanks to the parallel interest group (supervisors, Dave Sewry and George Wells) for the numerous discussions and support. Lastly, but not the least thanks to the staff of Rhodes Computer Science Department for the wonderful working environment.

ABSTRACT

With the evolution of parallel and distributed systems, users from diverse disciplines have looked to these systems as a solution to their ever increasing needs for computer processing resources. Because parallel processing systems currently require a high level of expertise to program, many researchers are investing effort into developing programming approaches which hide some of the difficulties of parallel programming from users. Linda, is one such parallel paradigm, which is intuitive to use, and which provides a high level decoupling between distributable components of parallel programs. In Linda, efficiency becomes a concern of the implementation rather than of the programmer.

There is a substantial overhead in implementing Linda, an inherently shared memory model, on a distributed system. This thesis describes the compile-time analysis of tuple space interactions which reduce the run-time matching costs, and permits the distribution of the tuple space data. A language independent module which partitions the tuple space data and suggests appropriate storage schemes for the partitions so as to optimise Linda operations is presented.

The thesis also discusses hiding the network topology from the user by automatically allocating Linda processes and tuple space partitions to nodes in the network of transputers. This is done by introducing a fast placement algorithm developed for Linda.

Table of Contents

List of Illustrations

Chapter 1
Introduction

1.1 Background

The advent of high speed VLSI CMOS chips has made low cost parallel processing now feasible. One such device, the **Transputer**, provides a uniquely simple component for constructing parallel machines. Since its inception it has proved to be reliable and flexible in building interesting systems for research into the behaviour of distributed memory **Multiple Instruction Multiple Data (MIMD)** machines [Fly66].

Given the availability of the hardware, attention has now focused on the remaining problems, foremost of which is the difficulty with which parallelism is specified. Currently under debate is the wisdom of developing new parallel languages as opposed to the modification of existing sequential languages. Much research is being done on the development of entirely new language models (i.e. models that are in themselves a complete language). Some researchers argue against the tremendous expense involved in developing, learning and utilizing a completely new paradigm [Ahu86], and maintain that this approach will only compound the problem.

To date, the needs of parallel programmers have not been accommodated very well. [Ahu86] isolates four uncompromising needs which most parallel languages do not cater for. These are

- A machine-independent and potentially portable programming vehicle.
- A programming tool that absolves them as fully as possible from dealing with spatial and temporal relationships among parallel processes.
- A programming tool that allows tasks to be dynamically distributed at run time.
- A programming tool that can be implemented efficiently on existing hardware.

One approach which may alleviate these problems is Linda, which is under investigation at Rhodes University and elsewhere. The programming style associated with Linda is elegant, simple, machine independent, and powerful [Dur89].

1.2 The Linda concept

Linda does away with the distinctions between synchronization, communication and process creation. Two fundamental notions associated with Linda are tuples and tuple space. A **tuple** is an ordered collection of typed fields, which is similar to a record in relational database theory [She90a]. **Tuple Space (TS)** is a collection of tuples. Communication between processes is achieved through the generation of data (passive)

tuples with a transfer of data from actual to formal tuple fields. The fields of **passive** tuples consist of a collection of data values which can be **actual** or **formal** parameters. An actual field parameter contains some physical value, while a formal field parameter is specified by a typed variable name preceded by a "?". Process creation is performed by generating live (active) tuples. The tuple fields of an **active** tuple consists of executing or executable code, which is regarded as occupying the TS environment.

A Linda process communicating with another generates and adds tuples to TS, while the other withdraws them. The communication is said to be **generative** because a tuple in TS has an independent existence. This tuple is equally accessible to all processes but is bound to none.

1.2.1 Linda (TS) Operators

There are 6 operators for accessing TS which can be incorporated into any sequential language to create the Linda dialect. The base language worries only about the computational part of the parallel program, while the Linda operators cater for the synchronization, communication and process creation aspects. These operators are:

out(tuple)

This operator inserts a tuple into TS. The executing process is not blocked. For example: out("results", row, col, sum) might place a tuple such as: ("results", 2, 5, 78.0) into TS.

in(template)

Using the tuple template, this operator extracts a matching tuple from TS. The process executing this operator gets blocked until a matched tuple is found and removed from TS. The conditions for matching are discussed in section 5.1. The actual parameters of the tuple are assigned to the formal parameters (denoted by a leading "?") of the template. For example: in("results", 2, ?j, ?ans) may remove the tuple sent in the previous example from TS. The variables j and ans will then be assigned the values 5 and 78.0.

If there are other tuples in TS which match the template supplied, then any one of the matching tuples can be chosen. The fields of the template need not include a formal. The operation in("results", 2, 5, 78.0) will remove the tuple ("results", 2, 5, 78.0) (or a similar tuple with formal parameters in any/all of the fields) from TS without any data transfer taking place. In a situation where the process blocks, all new tuples added to TS are checked until a match occurs.

rd(template)

This operator is similar to the *in* operator, except that the matched tuple is not removed from TS. Data is transferred to the process via the same formal/actual assignment mechanism.

inp(template) and **rdp**(template)

These are predicate variants of the *in* and *rd* operators, the difference being they do not block. They return a boolean variable to indicate whether or not the operation was successful. Data transfer occurs as before in the successful cases.

eval(tuple)

This operator inserts the *tuple* into TS. It creates multiple processes which can be executed in parallel with the process executing this operation. The tuple field values are evaluated in parallel, subsequent to entering TS. The result of evaluation is placed in the corresponding positions of the tuple. For example: eval("matrix", collect(sz,i,j), dot_product(i,j)) will evaluate the procedures collect and dot_product in parallel and the returned value of these procedures are placed in the tuple. An active tuple thus eventually becomes a passive tuple.

1.2.2 Advantages and Criticisms of Linda

A number of advantages of Linda discussed in the literature strongly support Linda as a general purpose parallel programming model. This section presents an overview of some of these.

- ***Ease of Use:*** Writing a parallel program in Linda is as easy as writing a sequential one due to Linda's characteristic uncoupled style [Ahu86]. Linda programs are written without considering:
  - the identities of the source and destination processors involved in communication.
  - the architecture underlying the application.
  - the synchronization and coordination of processes.

Linda programs are therefore written with concentration on the actual algorithms as opposed to implementation details.

- ***Orthogonality:*** Linda operators are orthogonal to the base language. As such, Linda can coexist peacefully with any number of base languages and computing models [Gel85].
- ***Portability:*** Linda programs can be ported from one architecture to another (irrespective of the processor model) without modification [Gel88].
- ***Scalability:*** Linda's programming model makes it possible to execute a program developed on a single processor on a multi-processor system with zero or minimal modification [Ahu86].
- ***Dynamic Load Balancing:*** Linda's programming model (replicated-worker) naturally distributes tasks between the available processors at run-time [Ahu86].
- ***General Development Tool*** - Linda is a good general purpose development tool [Gel88], in that Linda solutions are conceptually simple to understand and they exhibit real speed-up. This claim is supported by the wide range of application examples that Linda has been used for elsewhere [Gel85, Car89a, Car89b] and on our Linda system [Cla90].
- ***Power and Expressiveness*** - Linda is claimed to have greater power, expressiveness, simplicity and elegance

than any of the existing models of parallel processing paradigms [Gel85, Car89a].

Like any other parallel paradigm, Linda has its weaknesses. [Gel85] states its weaknesses stem from its strengths and reports the following:

- Programmers have to implement structures such as calling routines and queue management.
- It has no security mechanisms for preventing unauthorised access to TS.
- It presents difficult and novel implementation problems on distributed architectures.
- It can place massive communication overheads on network systems.

[Dav89] in reply to [Car89a] also cite the following criticisms:

- It has a massive run-time overhead.
- Tuple retrieval is a potential bottleneck.

1.3 Aims of this research

A tuple within TS is accessible by any node in the Linda system. This inherently makes Linda a shared-memory model [Ahu86]. But shared-memory machines are expensive when compared to distributed memory machines such as a network of transputers. Linda's unit of storage (a tuple) is sufficiently coarse to be implemented on distributed memory models. TS therefore need not be physically represented in shared memory: we can provide the necessary TS operations and use message passing mechanisms to implement the TS concept. This is the route that has been followed in the Linda implementation for a transputer network at Rhodes. To distinguish our implementation from the generic concept, we refer to this system as **Rhoda**.

Tuples do not have addresses, they are managed by a **TS Manager** [Wen89]. The easiest way of implementing TS on a distributed memory system is to centralize it. To locate a particular tuple we access the tuples in TS by their structure and contents of their fields as opposed to any form of physical address. This associative matching becomes expensive as TS grows, and the concentration of network traffic to one node can cause bottlenecks.

This research addresses the dual problems (criticisms) of expensive associative matching and the bottleneck caused by concentration of network traffic to one node. The thesis focuses separately on these issues, although they are not independent:

Firstly, the research tackles the **partitioning problem** by analyzing at compile time, a Rhoda program's TS interactions, in order to partition TS into non-overlapping subsets of tuples. These are called **tuple groups**. Appropriate storage schemes are then suggested for the tuple groups so as to reduce the run-time matching costs.

Secondly, the research attends to the **placement problem**. A fast heuristic algorithm is developed to statically place the different tuple groups and the Linda processes that access them, on transputer

nodes. The goal here is to balance the processor loads, while avoiding communication bottlenecks. It is noted that the success of placement depends on the decoupling provided by partitioning TS, and furthermore, doing the placement without the partitioning would enforce centralized tuple storage, which would not be very useful.

1.4 The scope of the research

The logical nature of a single shared TS has all the tuples for an application program stored at one node in an unfragmented memory space without any definite pattern. This makes associative matching quite expensive as a search is always directed to a single node and to the whole set of tuples. The central notion of this research is that search effort and/or network congestion can be reduced while balancing processor utilisation, by partitioning TS into tuple groups which may be individually placed and searched.

The placement algorithm assumes that apart from nodes controlling peripheral devices, the rest of the network consists of a homogenous set of transputers. It is also assumed that placement components to be replicated will be identified beforehand.

The TS partitioning module is language independent and therefore can interface with any conventional language's Linda pre-processor.

This research presents a partitioning algorithm which interfaces with the Rhoda pre-processor [Wel90], and a prototype of a placement algorithm. The main purpose of this latter effort was to clarify the placement issue for Rhoda application programs. This in turn will help clarify exactly what estimates of processing and traffic volumes are required from the compiler/pre-processor analysis. The placement is also expected to perform the initial allocation for a dynamic allocation scheme currently under investigation at Rhodes University.

The remainder of this thesis discusses parallel languages, Linda systems, and the contribution made by this research to the development of Rhoda. In chapter 2, fundamental concepts of parallel programming languages are discussed with emphasis on distinguishing features of Linda. We develop a classification scheme for Linda systems in chapter 3 and go on to survey a number of Linda implementations on IBM-PC compatibles, a local area network (LAN), and transputers. The Rhoda implementation is presented in chapter 4. We analyze the possible interactions within TS in chapter 5, on the basis of which we formulate the rules for partitioning TS. Using these rules we present and compare our two partitioning algorithms in chapter 6. Chapter 7 provides the background to process placements, and focuses more closely on the Rhoda requirements. The Rhoda placement algorithm is discussed in chapter 8, with chapter 9 evaluating the TS analysis module and the placement algorithm. Chapters 10 and 11 present future work and the conclusions of this thesis respectively.

Chapter 2
Fundamental Concepts of Parallel Languages

The first concurrent programming languages were designed with *shared memory-multi-processor* computer systems in mind, for example Concurrent Pascal [Bri75] and Modula [Wir77]. These languages were based on powerful mechanisms such as the monitor, semaphores and critical regions which provide structured access and synchronization in these systems. Although these mechanisms are very powerful, they are limited by their dependence on shared memory.

With the emergence of VLSI computing devices, cost-effective parallel processing hardware designs have tended towards multi-processor systems constructed from processors with own local memory, which communicate via a high speed network. The parallel languages designed with this target hardware in mind allow concurrent processes to communicate and synchronize with each other through some form of message passing. Having presented Linda in chapter 1, this chapter gives an overview of parallel programming models and explores the main distinguishing features of Linda to highlight some of the advantages discussed in section 1.2.2.

2.1 Models of parallel computation

The usual parallel programming paradigms for distributed memory computers include *explicit message passing*, *remote procedure call*, extending conventional sequential language, *declarative languages* and more recently *Linda*.

The *message passing model* is oriented towards the direct hardware connections found in multi-computers or networks of transputers rather than general networks. This approach offers a simple and fast low level communication between parallel processes. Occam 2, an example of this paradigm, is reviewed in the next section.

The *remote procedure call* paradigm promotes a client/server arrangement where the client is the program module that issues the call and the server is the module that handles the call. This paradigm provides a structured operation-oriented method with less explicit control, for example Ada [Ada83]. A number of distributed Ada programming environments have been developed, including a commercially viable one for transputer networks by Alsys. Ada is discussed in comparison with Linda in section 2.3.4. A number of other languages that use adaptations of remote procedure calls are *MOD (Star MOD) [Coo80], SR (Synchronizing Resources) [And82] and CP (Communication Ports) [Mao80].

Concurrent control constructs can be added to a popular sequential language to create a new language, for

example DISC (DIStributed C) [Ian89]. This method provides an alternative to using genuine parallel programming languages. An alternative method within this paradigm is to incorporate target-specific library functions which will provide a run-time transport layer. The disadvantage with this model is that programs are not easily ported to other execution environments.

Declarative languages (functional and logic) do not rely upon a particular order of computation. They obey the laws of substitution which makes the results of their computation independent of the order in which the sub-expressions are evaluated. This allows varying the order of evaluation so that it could be eager, lazy, or parallel. Two advantages of this paradigm are: they create new objects and hence repetitive references to shared information is greatly reduced, and they remove the need for explicit program partitioning by the programmer. Several documented proposals for implementing declarative languages on distributed hardware exist, for example Alice [Cri86].

2.2 CSP, Occam and the Transputer

The design of the Transputer and Occam are based on the principles of Communicating Sequential Processes (CSP) [Hoa78]. CSP is a formal system for reasoning about message passing parallelism. The main notion of CSP is to abstract over sequential parts of algorithms for the purposes of reasoning about communicating processes. It therefore facilitates proving deadlock-freeness in parallel systems.

The main vehicle for the specification of action in both Occam and CSP is the process. Occam and CSP have three primitive processes in common: input ("?"), output ("!"), and assignment (":="). In Occam, processes communicate by naming channels whereas CSP specifies the names of processes. For example

channel ? variable and channel ! variable

are Occam input and output statements respectively. The first receives a value on channel and assigns it to variable, while the second outputs a value onto channel. An output statement only terminates when there is input statement acting on the same channel. Similarly, an input statement requires an output statement to terminate.

The Occam constructor **SEQ** causes all statements within its scope to be executed sequentially, while the **PAR** constructor forces their execution in parallel. The corresponding CSP notations for these constructors are ";" and " || " respectively.

The basic architecture and instruction set of the transputer have been chosen such that Occam is normally the lowest programming level. Many functions of the run-time system are supported directly in hardware or firmware. Among these are message passing, process suspension, scheduling, queue manipulation and context switching. The next section compares Linda with other parallel paradigms using Occam and Ada for illustration.

2.3 Some Distinguishing Properties of Linda

Linda differs significantly from shared memory programming, message passing, remote procedure calls and functional languages. Although Linda belongs to the class of extended conventional sequential languages, it differs from other parallel processing languages in this class. Linda processes do not communicate directly with each other and it should be recalled from section 1.2.2 that Linda programs are easily ported from one environment to another.

Linda operators were described earlier as being orthogonal to the base language in which they are embedded. This gives it a unique communication feature which [Gel85] refers to as *Communication Orthogonality*. In other distributed programming languages, the send statements name their receive process explicitly in some fashion. Linda does not. For example in Ada:

```
task Client;

task body Client is
  ...
  Server.call(x, y);
  ...
end Client;
```

```
task Server is
  entry call;
end Server;
task body Server is
  ...
  accept call(x:item; y:out item) do
  ...
end call;
...
end Server;
```

In this remote procedure call model, the caller must name the callee, but the callee has no knowledge about the caller. Similarly in the message passing model of Occam the caller must name callee and the callee must also name the caller (implicitly by the declaration of a shared point-to-point channel). For example:

```
PROC Client(CHAN OF INT call, reply)
  SEQ
    ...
    call ! x
    ...
    reply ? y
    ...
:
```

```
PROC Server(CHAN OF INT call, reply)
  SEQ
    ...
    call ? x
    ...
    reply ! y
    ...
:
```

However, in the Linda model, the receive statement has no prior knowledge about the sender, and the send statement also has no knowledge about the receiver. The send statement may therefore send to a number

of other processes and the receive statement may receive from a number of other processes.

This anonymous communication style is said to *Spatially decouple* Linda processes. This differs from Ada, where processes communicating are coupled by call statements, and Occam where they are coupled by channels.

A further implication of the orthogonality of Linda operators and of TS is *temporal decoupling.* A Linda process executing a send statement may run to completion before the process executing the corresponding receive statement is loaded. In models such as Ada and Occam, the communicating processes must be executed concurrently.

Linda allows a number of processes to share a variable (*distributed sharing*) by maintaining it atomically in TS. Other parallel languages will require a module or a process to implement this [Gel85]. The next section presents a sample Rhoda program.

2.3.1 An Example of the use of Linda

To illustrate programming in Linda, a matrix multiplication program for the Rhoda system is presented below. This example employs the **replicated worker** model. In this model, the master process outputs requests for work (computing a row of the product of two matrices) into TS. These requests are picked up by worker processes which perform the required computation and then place their results back into TS to be collected by the master process.

```
/* Master process */

%%
unsigned int endTime, startTime;
double *A, *B;

int rmain(int argc, char **argv)
/*====================*/
{

  clrscr();
  printf("Linda matrix multiplication by rows\r\n");
  matrix_demo();

  RestoreConsole();
  return(0);
}

void matrix_demo(void)
/********************/
{int sz;

   printf("\nSquare size of matrix? ");
   fflush(stdout);
   scanf(" %i", &sz);
   startTime = clock();

   out("size", sz);
```

```
/* Worker process */


%% mat_wrkr

int rmain (int argc, char ** argv)
/*=====================*/
{ double *R1,*R2,*R3;
  int sz, dummy;

  in("size", ?sz);

  R2 = malloc(sz*sz*sizeof(double));
  rd("matrix", ?R2:dummy);

  R1 = malloc(sz*sizeof(double));
  R3 = malloc(sz*sizeof(double));

  if (R1 == NULL || R3 == NULL)
  { fprintf(stderr,"Not enough memory\n");
    exit(1);
  }

  { int bytesz, i, row, col;
    double sum;

    while (1)
    {
```

```
    A = malloc(sz*sz*sizeof(double));
    fill_matrix(A,sz);
    out("matrix", A:sz*sz*sizeof(double));

    distribute_rows(A, sz);
    eval(mat_wrkr);
    collect_rows(A,sz);
    endTime = clock();
    gotoxy(1,23);
    printf("\nTotal time = %2.1fsecs.\r\n",
      (endTime-startTime)/ (float) CLK_TCK);
  }

void fill_matrix(double A[], int sz)
/*=================*/
{ int i,j;
  double *p;
  p = A;
  for (i=0; i<sz; i++)
    for (j=0; j<sz; j++)
      *p++ = (i+j) % 2;
}

void distribute_rows(double A[], int sz)
/*==================*/
{ int i,count;
  double *p;

  printf("Distributing matrix\r\n");
  p = A;
  count = sz * sizeof(double);
  for (i = 0; i<sz; i++)
  {
    out("rows", i, p:count);
    p += sz;
  }
}

void collect_rows (double B[], int sz)
{ int i,k,num_bytes,psz,count;
  double *R3;
  R3 = malloc(sz*sizeof(double));

  clrscr();
  count = 0;
  gotoxy(52,2); printf("Rows collected = ");
  psz = (sz > 12 ? 12 : sz);
  for (k=0; k<sz; k++)
  { in("results", ?i, ?R3:num_bytes);
    memcpy(&B[i*sz],R3,num_bytes);
    if (i < 12)
    { gotoxy(1,i+2);
      for (i=0; i<psz; i++)
        printf("%6.2f",R3[i]);
    }
    count++;
    gotoxy(72,2); printf("%6d",count);
    fflush(stdout);
  }

}
```

```
    in("rows", ?row, ?R1:bytesz);

    for (col=0; col<sz; col++)
    { sum = 0.0;
      for (i=0; i<sz; i++)
        sum += R1[i]* (*(R2+i+sz*col));
      R3[col] = sum;
    }
    out("results", row, R3:bytesz);
  }
}

  return(0);
}
```

2.4 Summary

A survey of parallel computation paradigms other than Linda have been presented in this chapter. Linda has

also been compared to Ada and Occam to point out the distinguishing features which makes it easy to use. These features are communication orthogonality, spatial and temporal decoupling of Linda processes, and distributed sharing.

Chapter 3
Linda Implementation

Implementing tuple space on a machine that lacks physically shared memory poses a number of implementation problems [Gel85,Car87]. One issue that has to be addressed carefully is how to provide effective access to the tuples in tuple space. This in turn raises the principal question of how to store tuples on distributed memory systems. A number of TS implementations on distributed memory systems are currently under investigation in a number of research institutes. To investigate the different aspects of these implementations independently, this chapter suggests a possible classification scheme. Figure 1 illustrates the structure of a general Linda system.

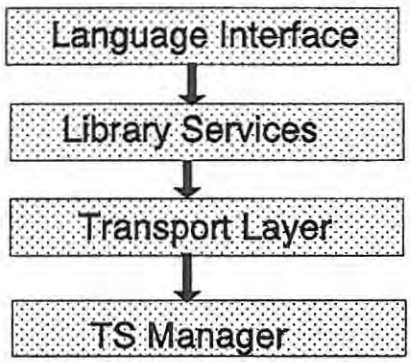


Figure 1. A general Linda system implementation.

TS implementation characteristics (storage, and data transfer) have much in common with distributed database systems. A number of database classification schemes were considered so as to find a suitable classification scheme. The overview of a few schemes encountered in the literature is presented below.

In discussing the pros and cons of centralized versus decentralized computing, [Kin83] classifies databases as Centralized (those that are managed on a central storage), Intermediate (those that are consolidated onto one or few centres), or Decentralized (those that are spread over a number of centres).

[Kim84] examines typical machine and storage organizations for database systems. He classifies multi-processor systems as loosely coupled (processors execute their own copy of the operating system in their own main memory) or tightly coupled (processors execute the same copy of the operating system in a common shared memory). Database systems are then classified as:

- tightly coupled with a shared database (centralized database).
- loosely coupled with a partitioned database (partitioned database).
- loosely coupled with a shared database (centralized database).
- loosely coupled with a replicated database (replicated database).

[Jar84] discussing query optimization in database systems, classifies databases physically as centralized or distributed. He further classifies distributed databases as:

- replicated (data replicated at all centres for high availability),
- local storage (data stored at their respective centres for improved accessibility by frequent users), and
- data fragments with/without overlaps.

In reviewing programming languages for distributed computing systems, [Bal89] defines distributed data structures as data that can be manipulated simultaneously by several processes. They then isolate Linda systems that:

- replicate the entire TS on all processors.
- hash tuples onto specific processors.
- store tuples on the nodes where they are created.

[Ada89] examines security-control methods for statistical databases. They classify databases as centralized or decentralized (distributed) with overlapping subsets which maybe:

- fully replicated,
- partially replicated, or
- partitioned.

[Elm90, Lit90, She90b, Tho90] address the issue of Heterogeneous and Autonomous databases. They all classify databases as centralized or distributed (decentralized).

None of the classification schemes encountered in the above literature and others exhaustively cater for Linda's TS implementations. For example, none caters for storing data as tuple groups. Consequently, we have developed a new classification scheme using the basis of the schemes above to investigate Linda systems. The next section introduces our scheme.

3.1 A Linda classification Scheme

We have adapted Quinn's classification of multi-processor systems [Qui88] to physically classify the architecture for the Linda system. Linda system architectures can therefore be loosely coupled (shared address space formed by combining the local memories of the processors), tightly coupled (processors work through a central switching mechanism to access a shared global memory), or a multi-computer (processors have own private memory, and all communication and synchronization done via messages). The topology of these architectures can either be fixed (processors connected in a rigid pattern), or arbitrary (processors connections not bound by any set pattern). For our logical classification scheme, we restrict ourselves to

multi-computers.

At one extreme of the Linda implementation spectrum, all tuples inserted into TS are stored on a unique node in the system. This we term the **centralized TS**. This has the advantages of:

- economical memory utilization [Bjo87].
- ease of TS consistency maintenance.
- a fast *out* operation [Gel85].

The disadvantage of this implementation is:

- in the worst cases, the distance that a message must travel between nodes will cause severe communication overheads and bottlenecks.
- processor overloading if many requests arrive in a short space of time.

A centralized TS may be **partitioned** (tuples partitioned into non-overlapping groups and stored as tuple groups) or **hashed** (tuples hashed and stored as a table).

At the other extreme of the spectrum, TS is spread over a number of nodes in the system. This implementation is referred to as a **distributed TS**. Under distributed TS, a number of strategies can be employed to distribute and retrieve tuples.

In one approach, tuples injected into TS are passed to and stored on every node in the system. This implementation is termed the **replicated TS**. The advantage of this implementation is:

- fast *out* and *rd* access [Wen89].

The disadvantages are

- uneconomical memory utilization [Bjo87].
- slow *in* access, because of the need to ensure exclusive deletions [Car87].
- difficulty of maintaining TS consistency [Car87].
- heavy traffic penalties in some networks especially point-to-point ones.

A replicated TS may also be partitioned (tuples on each node partitioned and stored as tuple groups) or hashed (tuples hashed and stored as a table on each node).

Another scheme stores tuples on the nodes where they are produced. This is referred to as a **localized TS** implementation. This scheme does not replicate any tuple and it has the advantages of a centralized TS and,

- immediate availability of locally produced tuples.
- tuples are not moved about in the network unless they are needed elsewhere.
- in situations where a match is always guaranteed, this approach reduces the traffic overhead compared to the centralized scheme.

The disadvantages are:

- it introduces additional search overhead for the network for tuple requests that cannot be satisfied locally.
- need to ensure that local and network updates of TS are mutually exclusive.

An alternative strategy is to partition the tuples into tuple groups. These tuple groups can then be stored on unique nodes. This implementation is referred to as the **partitioned TS.** This method has the advantages and the disadvantages of a centralized TS but it is more flexible in that communication distances can be reduced by judicious placement of the tuple groups.

In a system where a key field always exists within a tuple, tuples can be hashed onto specific nodes. This implementation is referred to as a **hashed TS.** This scheme also has the advantages and disadvantages of the centralized TS.

There are a number of distributed Linda systems which use special fixed topologies. Such systems are referred to by their fixed topology. The overview of two such implementations are discussed in sections 3.4 and 3.5.

The remainder of this chapter is devoted to a review of some Linda systems.

3.2 A replicated TS implementation on a Network of IBM-PC compatibles

An earlier Linda implementation at Rhodes University, running on a ring of PCs interconnected through RS-232 ports [Wen89], implements a replicated TS. This system is defined in three distinct layers, the *Language Interface*, the *TS Manager* and the *Transport Layer.*

The Transport layer routes local TS changes on a PC to other PCs in the ring. A PC acting as the ring leader assigns processor-ids to each node in the ring, generates a token which determines exclusive delete access to TS, inserts timing delays in its circulation and shuts down the system. In this system, maintaining TS consistency is ensured by deleting tuples atomically on all PCs. Each delete message is circulated to all nodes in the ring, and all pending delete messages on a PC are sent before passing the token. In addition to maintaining TS consistency, this token mechanism helps to reduce the network traffic.

The TS manager implements a hashed TS for each PC. Search routines can reproduce the hash information from the request. In the case of an *in* request, the TS manager acquires the delete token, deletes the tuple, reclaims the memory and calls the transport layer to send a deletion request to the other PCs.

The language interface defines the client's view and interface to Linda. This syntactically sugared interface

is translated into direct calls to the TS manager. [Wen89] describes this system in detail. Figure 2 illustrates this implementation.

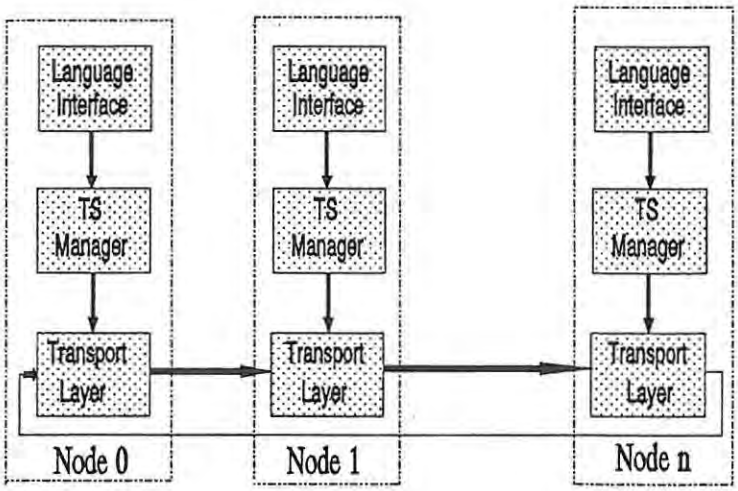


Figure 2. A replicated TS implementation on IBM-PC compatibles

3.3 A localized TS implementation on a Network of IBM-PC compatibles

A second PC Linda system developed at Rhodes University, implements a localized TS. This makes use of a Local Area network (LAN) [Dan90], and also has three levels, a Language Interface, a Local TS device driver and a NetBIOS[1] transport layer.

This system implements the Linda operators (in, rd, out) as a Linda device [Dan90]. By providing a DOS device for the Linda device, it provides a language independent application interface. Tuples that are produced are stored locally. To address the additional search overhead for tuple requests that cannot be satisfied locally, a broadcast message is sent to the other PCs in the network. A locking mechanism on each node ensures the local application and the network server do not update TS simultaneously. A detailed description of this implementation is given in [Dan90].

3.4 TS implementation on a grid of transputers.

This system needs a fixed topology, and has been implemented on a $\sqrt{k}$ by $\sqrt{k}$ grid of transputers as illustrated in figure 3 below. This system employs an intermediate uniform distribution [Ahu86] of tuples. It uses the concepts of tuple beams and inverse tuple beams (referred to by [Gel85] as **out-threads** and **in-threads**).

[1]. Software interface to low-level network functions. Network Basic Input/Output System.

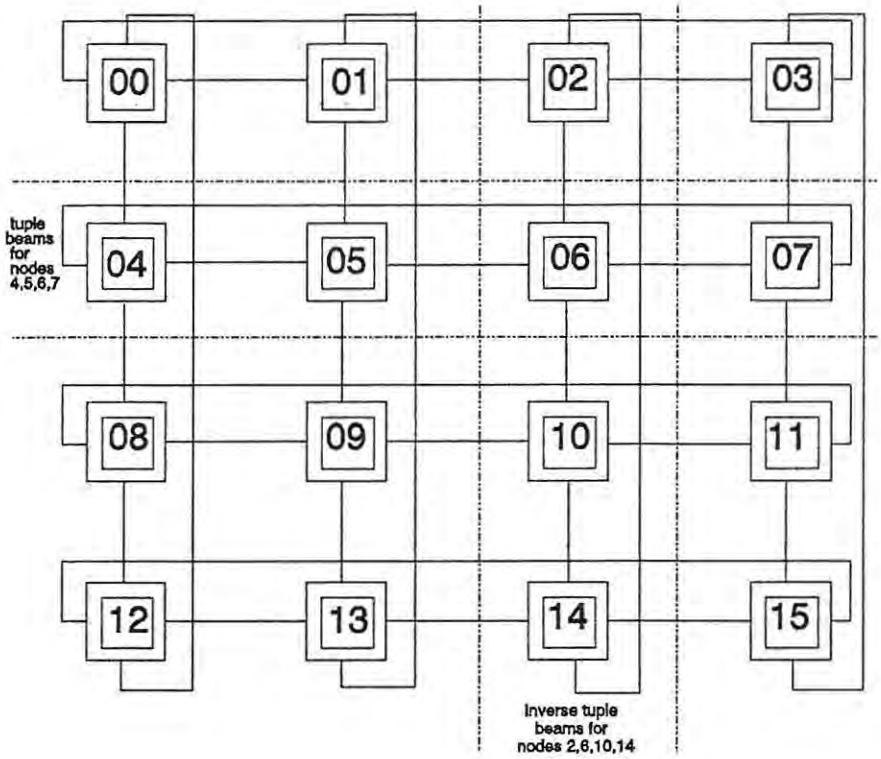


Figure 3. TS implementation on a grid architecture of transputers (figure extracted from [Faa90])

A tuple produced by a node is sent to all nodes on its row (tuple beam). A row of the grid is thus a replicated TS. A request for a tuple on any node is sent to all nodes along its column (inverse tuple beam). Each column of this scheme can be viewed as a localized TS. For a grid of k nodes, a tuple beam and inverse tuple beam both comprise of $\sqrt{k}$ nodes which intersect. Since the number of nodes involved in the insertion and withdrawal of a tuple is only $2\sqrt{k}$, this scheme is quite scalable and may be favourable for larger networks.

A tuple match occurs on a node if the node is in possession of both the template and a matching tuple. To maintain TS consistency, nodes producing tuples contend for the out-bus, while nodes withdrawing tuples from the same in-thread, contend for the in-bus and then the out-bus to retrieve and delete the tuple. Nodes on different in-threads first attain ownership of the out-bus.

On one such implementation at the University of Witwatersrand [Faa90], the Linda primitives *in*, *out*, and *rd* are implemented in Occam on a 16-processor SuperCluster. A detailed description of this implementation is given in [Faa90].

3.5 Distributed TS ring implementation on Transputers

A Linda system proposed at Yale University also needs a fixed topology. TS is implemented as a separate subsystem on a ring of dedicated transputers. To route messages, the ring is configured as two unidirectional rings, one travelling clockwise, the other anti-clockwise. Two computation nodes are attached to each node in the ring. This architecture, [Zen90] claims, is scalable up to interesting sizes, and that as the ring size escalates, short circuit nodes can be added, with minimum cost, to keep the mean distance messages travel to some constant. The figure 4 below illustrates the architecture for this implementation.

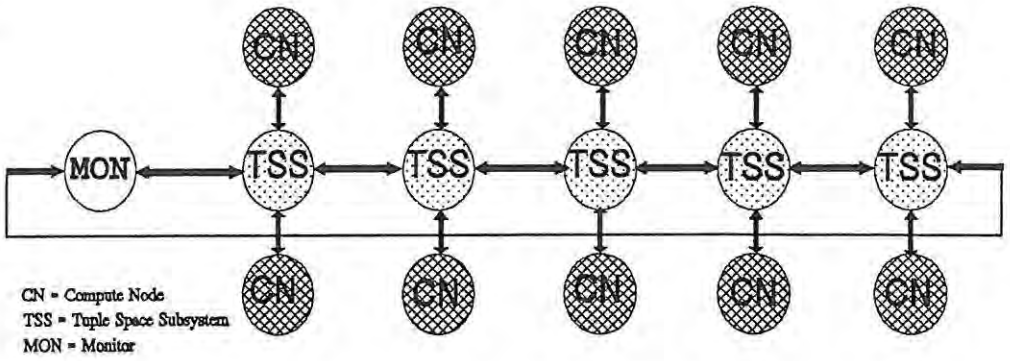


Figure 4. A distributed TS ring implementation on transputers (Extracted from [Zen90])

This system also implements a partitioned TS. The tuple groups are distributed evenly among the available processors in the TS subsystem using a hashing scheme. A detail description of this implementation is given in [Zen90].

3.6 Summary

In this chapter we have developed a classification scheme for Linda implementations on multi-computers. We broadly classify the logical systems as:

- a centralized TS which stores all tuples on a single node. or
- a distributed TS where tuples are distributed over a number of nodes.

A distributed TS is further classified as:

- a replicated TS which stores a tuple on every node in the system.
- a hashed TS, in which a tuple is hashed onto a unique node.
- localized TS, which stores tuples on the nodes where they are produced.
- partitioned TS, which partitions tuples into tuple groups and stores each group on a unique node.

This scheme has been used to examine and classify four Linda systems. The next chapter presents an overview of the Rhoda system.

Chapter 4
The Rhoda Implementation

The Rhoda system [Cla90], unlike the two Linda systems (on a grid and ring of transputers) reviewed in chapter 3, has been implemented to run on arbitrary transputer topologies. Rhoda implements a partitioned TS.

The ideal syntax of Rhoda provides a high level decoupling of Rhoda distributable components. One of the disadvantages of the high level decoupling of Linda is that efficiency becomes the concern of the implementation rather than that of the programmer. This can be viewed as an advantage since the programmer is freed from efficiency concerns. This chapter describes the Rhoda system, which is the primary target of this research.

4.1 Rhoda System Overview

Rhoda is defined as a four part system and runs under the Helios operating system (appendix 1). These are the **language interface, concrete interface, transport layer** and the **TS server**. Figure 5 below illustrates the structure of the Rhoda system.

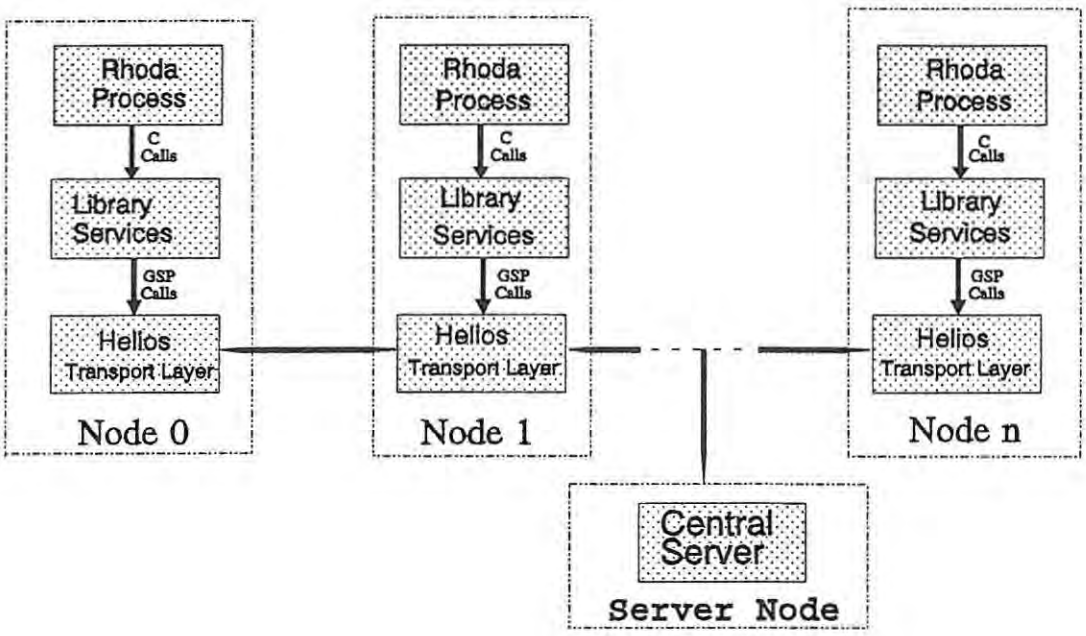


Figure 5. The Rhoda implementation

Although a partitioned TS is in contrast to the Linda paradigm of a single shared TS, it is the Rhoda

compiler which adds additional housekeeping information to source programs, transforming them to work with this partitioned TS model. Figure 6 illustrates the Rhoda compilation path.

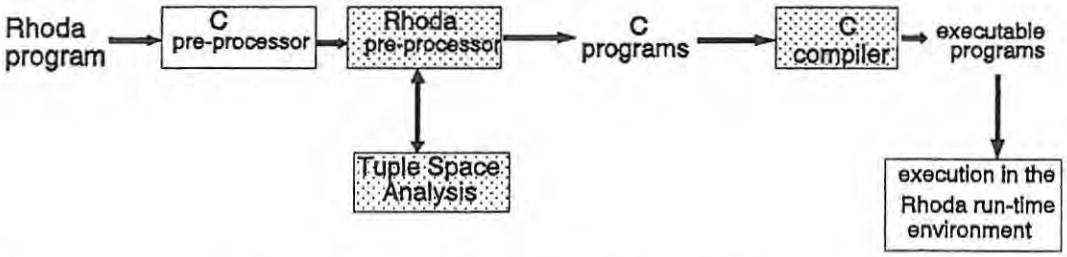


Figure 6. Structure of the Rhoda compilation path

The current host language for the Rhoda system is C. The compilation is a four-pass system. The Rhoda program is passed through the Rhoda pre-processor, which extracts all Rhoda operations. This list is fed to the analysis module, which has been written as part of this research. This module analyzes TS interactions with the application program, so as to partition TS and to suggest storage schemes for the tuple groups. The grouping of tuples is thus an integral feature of the Rhoda implementation, and not a concern of the application programmer.

The Rhoda pre-processor uses the output of the TS analysis to translate the ideal syntax into standard C syntax calls to the library functions. This is called the *concrete C syntax*. The output of the Rhoda pre-processor is a series of C programs - one for each program component. These C programs are then compiled by the C compiler to produce executable programs for the run-time Rhoda environment.

Rhoda supports character strings, integers, real (or floating point) values and a **byte block** (an arbitrary block of data which can be used to transfer arrays, records or other complex data types). The language interface (ideal syntax) has been covered in section 2.3.2. The remaining sections of this chapter deal with the concrete interface, the transport layer, and the TS server.

4.2 The Concrete Interface

Rhoda application programs interface with the concrete interface (library services) via normal C calls, and run on the same processor and memory space. These calls are converted into GSP (general server protocols) mechanisms which makes data passage through the network transparent to processes. Statistical information is also provided to a client process wishing to monitor TS.

To illustrate the concrete C syntax, the worker process of the matrix multiplication program after transformation is given below.

```
int rmain (int argc, char ** argv)
{ double *R1,*R2,*R3;
```

```
  int sz, dummy;

IN(TG_matrow[2], 0, &sz );
  R2 = malloc(sz*sz*sizeof(double));

  READ(TG_matrow[3], 0, &dummy , R2 );

  R1 = malloc(sz*sizeof(double));
  R3 = malloc(sz*sizeof(double));
  if (R1 == NULL || R3 == NULL)
  { fprintf(stderr,"Not enough memory\n");
    exit(1);
  }

  { int bytesz, i, row, col;
    double sum;
    while (1)
    {
      IN(TG_matrow[0], 0, &row , &bytesz , R1 );
      for (col=0; col<sz; col++)
      { sum = 0.0;
        for (i=0; i<sz; i++)
          sum += R1[i]* (*(R2+i+sz*col));
        R3[col] = sum;
      }
  OUT(TG_matrow[1], row , bytesz , R3 );
    }
  }
  return(0);
}

void main (int argc, char * argv[])
{
  /* Tuple group open operations */
  TG_matrow[0] = OpenTS("matrow0", "ib", M_Passive, 1024, malloc);
  TG_matrow[1] = OpenTS("matrow1", "ib", M_Passive, 1024, malloc);
  TG_matrow[2] = OpenTS("matrow2", "i", M_Passive, 1024, malloc);
  TG_matrow[3] = OpenTS("matrow3", "b", M_Passive, 1024, malloc);
  TG_matrow[4] = OpenTS("mat_wrkr", "", M_Active, 1024, malloc);

  while (POLLP(TG_matrow[4]))
    rmain(argc, argv);

  /* Tuple group close operations */
  CloseTS(TG_matrow[0]);
  CloseTS(TG_matrow[1]);
  CloseTS(TG_matrow[2]);
  CloseTS(TG_matrow[3]);
  CloseTS(TG_matrow[4]);
} /* main function for Rhoda section 2 (mat_wrkr) */
```

By this stage, the TS partition analysis is responsible for a number of transformations in the Rhoda ideal syntax. TS has been partitioned into five tuple groups. Each of these groups is declared, opened and closed after use by program components. The open tuple group command includes a specification of the types of each field, and the sizes of the transport buffers. Furthermore, the common literal fields of each tuple group have been discarded and each Rhoda operation specifies the filename of the tuple group of interest.

4.3 The Transport layer

The Transport layer is also handled by Helios GSP mechanisms. The processor manager spawns an IOC (I/O

controller) process for each task on a processor. This IOC process acts as the task's communication intermediary with the rest of the system. Each physical link of a processor also has an IOC responsible for handling distributed searches and requests from remote tasks to local servers. The IOCs on one processor route requests to named objects on behalf of their tasks by referencing the processor's *name table*. If the object is present in the table, the IOC passes the request directly to the server whose port is represented in the entry. If not, a distributed search is initiated, and provided the named server exists elsewhere, an entry is installed in the name table. Subsequent requests are then routed directly.

4.4 The TS server

The Rhoda TS server is implemented as a Helios server, using standard GSP protocols. The generic utilities which operate on other Helios objects (e.g. 'ls' to list directories) are able to operate on TS structures as well. Each TS server controls one or more tuple groups.

The first reference of a Rhoda process to a tuple group initiates a dynamic network search and establishes a connection path between the client component and its proxy process (appendix 1) in the remote server. Thereafter the client process has a point-to-point virtual link to the dedicated proxy process, which manipulates the tuple group on its behalf, until it requests a close operation. This makes it possible for the two to exchange messages without regard to the system topology.

The TS proxy processes assume the responsibility for locking the tuple group and coordinating requests during operations which update the group. A TS proxy server handles an unmatched request by queuing it, along with a semaphore, in the waiting queue for the tuple group it supports. The proxy process then suspends itself by waiting on the semaphore. Because the proxy has not yet replied to the client's request, the client also becomes suspended, awaiting the reply. Each time a new tuple arrives for the group, a pass is made through the queue to locate each matching *rd* transaction which can be completed. The first matching *in* transaction consumes the tuple. When each match is found, the proxy is re-activated and the data is returned to the client. If the tuple is not consumed by an *in* transaction, it is added to the tuple group in the normal way.

In a Rhoda application program, the relationship between processes and tuple groups is *n:m*.

4.5 Summary

This chapter has presented an overview of the Rhoda system design, with special attention to the layers of abstraction, the compilation path, and the interaction between Helios and the transport layer and the server. The user has been isolated from the concrete syntax by provision of a Rhoda pre-processor. At the concrete level, a library supporting server-client arrangement is used to provide TS services to the user. The Helios GSP mechanisms hide the network topology and routing concerns from the library code, and provide a very

general transport layer that allows clients and servers to be distributed. The remaining chapters of the thesis concentrate on the TS analysis and components placement.

Chapter 5
Tuple Space Interactions

In this chapter, we consider the run-time TS interactions which will be used as the basis to optimize Rhoda operations at compile time. We also consider those interactions that make it possible to implement the shared memory model of TS as a distributed system. The compile time analysis involved is presented in two stages, the partition analysis and the matching analysis. Recall from section 4.1 and 4.2 that the Rhoda pre-processor must transform the ideal syntax of Rhoda application programs to concrete syntax which works with file-like partitions. The partition analysis produces these partitions which in turn make it possible to implement a distributed TS system. The matching analysis explores strategies for storing individual partitions.

5.1 Rules for Matching Tuples

There are basically four rules governing the matching of a template against a tuple in TS at run-time. These are

- the field types must match.
- actual field parameters must be the same.
- formals in a template always match actual fields in a tuple.
- formal fields in a tuple can only match actual fields in a template.

5.2 Conditions for a match to fail.

In this section, we identify the instances where a tuple in TS fails to match a template at run-time. These instances provide a key to partitioning TS at compile time.

Linda operators can be classified as consumer operators and producer operators. The operators for withdrawing a tuple from TS, *in* and *inp*, and those for looking at a tuple, *rd* and *rdp* will collectively be called the *"input" (consumer) operators*. The operators for putting a tuple into TS, *out* and *eval*, will also collectively be referred to as the *"output" (producer) operators*. In this section, the *in* operator will be used to illustrate a TS "input" operation in the discussion below. The tuple and template field types will be designated as follows, *s* for a character string, *i* an integer, *r* a real, and *b* a byte block in the run-time TS interactions discussed below.

case 1:

Tuples and templates with distinct type signatures cannot match. For example, let the following tuples be in TS,

("row", i, j, result) with type signature <siir>,

("row", r, c, result) with type signature <sbbr>,

and let the following be Rhoda operations.

in("row", ?i, ?j, ?res) with type signature <siir>,

in("row", ?k, ?c, ?res) with type signature <sbbr>.

From the rules of tuple matching, the tuple ("row", i, j, result) cannot match the template ("row", ?k, ?c, ?res), because i is an integer, but k is not an integer, and moreover c is also not of the same type as j. Similarly the tuple ("row", k, c, result) cannot match the template ("row", ?i, ?j, ?res). This suggests that, tuples ("row", i, j, result) and ("row", r, c, result) may be placed in different tuple groups. The operation in("row", ?i, ?j, ?res) can then be directed to the first group, while in("row", ?k, ?c, ?res) can be directed to the other.

For the next two cases of this discussion we assume that all tuples and templates have identical type signatures.

case 2:

A tuple and template with distinct literal values in identical field positions cannot match. For example, let the following be tuples in TS.

("row", i, j) with type signature <sii>,

("col", i, j) with type signature <sii>,

and let the following be Rhoda operations:

in("row", ?i, ?j) with type signature <sii>,

in("col", ?i, ?j) with type signature <sii>.

The template ("row", ?i, ?j) cannot match the tuple ("col", i, j), because the actual field value "col", is not the same as "row" in the template. This also suggests that the two tuples may be placed in different tuple groups provided there is no "input" operation with a template having a wildcard in the corresponding field of the literal. The "*input*" operations templates can then be directed to these groups accordingly.

case 3:

A tuple with a formal field parameter cannot match a template with an corresponding formal field parameter. For example, let the following be two sets of tuples in TS.

Set 1	Set 2
(?y)	(?a, B)
("z")	(A, ?b)

and let the following be Rhoda operations:

in(?x) in(A, ?b)

in("u") in(?a, B)

In the first set, the tuple (?y) can only match the template ("u"), and tuple ("z") can only match template (?x). These tuples can therefore be assigned to individual groups, and the "input" operations directed to the relevant groups. Similarly, the second set can be treated likewise.

5.3 Partition Analysis

Recall from section 1.3, the partitioning of TS is supposed to be a compile time job. The necessary information about tuples that will be in TS at run-time can be obtained by examining the templates of the Rhoda "output" operations. This makes it possible to examine the Rhoda operations in an application program to partition TS at compile time. Section 6.2 presents the two algorithms that partition TS.

5.3.1 Constant Tuple fields

A tuple group is essentially an anonymous relation (collection of tuples). Where all tuples and templates carry an identifying string, this can be regarded as a self-identifying tag which names a specific relation: once those tuples and templates are grouped into their own partition, the "*relation name*" can be factored out and thrown away. Their function is now replaced by the selection of the tuple group. For example, operations which refer to a group of tuples such as

("row", i, j) ("row", k, l) ("row", ?m, ?n)

can be modified to operations on the same "*row*" tuple group using the tuples

(i, j) (k, l) (?m, ?n)

respectively. This reduces the length of tuples and templates, and hence run-time matching as well as the transport buffer size.

5.3.2 Tuple Groups Usage

It is possible to take the partition analysis further by determining how each process of an application program uses each of the tuple groups. The pre-processor can then determine which tuple groups to open for each program component thus preventing a process from accessing a tuple group it does not use. This usage information also provides information which is useful for the placement of tuple groups relative to the processes which they serve in the processor network.

An examination of the operations performed on each tuple group makes it possible to report some programming errors. For example, if the templates in a tuple group are used in only "*output*" or only "*input*" operations, this indicates a programming error and the programmer should be alerted.

5.3.3 Hierarchical TS

A partitioned TS allows an hierarchical naming scheme for a centralized TS implementation. For example, let an application program comprise three processes, P_1, P_2 and P_3, and let them coordinate their parallel activities through the use of three tuple groups. Suppose all three processes make use of tuple groups 1 and 2, while only P_2 and P_3 make use of tuple group 3. Figure 7 illustrates the hierarchical relationship which results from partitioning TS.

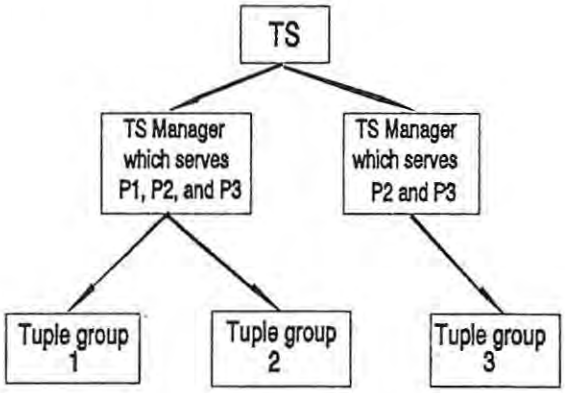


Figure 7. One possible hierarchical decomposition of TS

The Helios hierarchical naming scheme for network objects can therefore utilise this in a TS implementation.

5.4 Matching Analysis

In addition to directing each Rhoda operation to a specific tuple group, the run-time retrieval of tuples from TS can be optimised further. The TS manager can organise efficient storage schemes for individual tuple groups, so as to provide efficient run-time search strategies within each tuple group. The polarity (actual or formal) of the remaining fields can be used to determine these storage schemes.

The possible field patterns for the templates in a group after discarding common literal fields are as follows:

- a. templates with no fields.
- b. "input" template fields are all formal while "output" template fields are all actual.
- c. "output" template fields are all formal while "input" template fields are all actual.
- d. An actual parameter exists in a particular field position for all tuples and templates.
- e. No pattern in the field parameters.

Tuples in a tuple group with case *a* are being used to synchronize the activities of processes. In such an instance, the tuple group can be implemented as a counting semaphore and in so doing eliminate run-time matching.

For case *b*, any tuple in TS will match the template supplied in an "input" operation and perform a transfer

of data from actual to formal fields. The group can be implemented as a simple queue with no run-time matching.

Similarly in case *c*, no run-time matching will be required, and because no data transfer takes place, the TS manager can implement such a group as a counting semaphore.

For any instance of case *d* in a group, the TS manager can use the field as a key to store that group as a hash table. If there are more than one key fields, they can be used together to compute a hash key. The correct hash slot during a match can be selected by hashing the key field/fields in the template. Besides, if the other fields of the "*input*" template are all formal, then the first value found in the hash slot will always match and can be returned without further checking.

If we are not so fortunate and the analysis reveals pattern *e*, such an instance may necessitate an exhaustive search at times. The TS manager may implement this tuple group as simple queue and exhaustively search the list until a template-tuple match is successful. Although this usage of tuples is idiosyncratic, after partitioning and discarding common literal fields, it becomes a more common case. The Rhoda system implements such a tuple group as a simple queue.

5.5 Distributing TS

After partitioning, it is possible to derive a binary communication matrix of the placement components of the application program. This matrix shows which components communicate directly with each other. This binary matrix can then be used by the cost pre-processor to estimate initial costs for the placement module.

In a replicated TS system, where each node keeps copies of only the tuple groups used by the processes running on it, the Transport Layer can use the tuple group usage analysis to route local TS changes to only the relevant nodes. This can reduce the network traffic in the system.

In a distributed TS system, unique distribution of TS groups may not be enough for load balancing. The replication of certain tuple groups within TS might be a further alternative. The difficulties of maintaining TS consistency and ensuring unique deletion of tuples for a replicated TS will arise. Where tuple group usage analysis determines that certain processes only read, but never delete tuples, this information will enable these tuples to be replicated without having to take these factors into consideration.

5.6 Summary

This chapter has considered the run-time TS interactions and shown what conditions are necessary to optimize Linda operations and implementations. The conditions surveyed are the partition analysis, the matching analysis and their possible uses in optimizing a Linda system. In the next chapter, we consider the

implementation of some of these ideas.

Chapter 6
The TS Analysis Module

This module implements the partition analysis and the matching analysis discussed in chapter 5. The module is split into three phases. These are the Rhoda pre-processor interface, the partition of tuples into tuple groups, and derivation of storage schemes and usage information of tuple groups. This chapter considers each of these in turn.

6.1 Rhoda Pre-processor Interface

The TS analysis module interfaces with the Rhoda pre-processor through textfile input and output. This makes the module language independent and it can therefore interface with any other Linda pre-processor. An output file from the Rhoda preprocessor during its first pass on a Rhoda application program serves as the input file. The analysis module in turn outputs its results which serve as input to the Rhoda pre-processor during its second pass. It is during this pass that the pre-processor transforms the Rhoda operations written in ideal syntax into concrete C syntax.

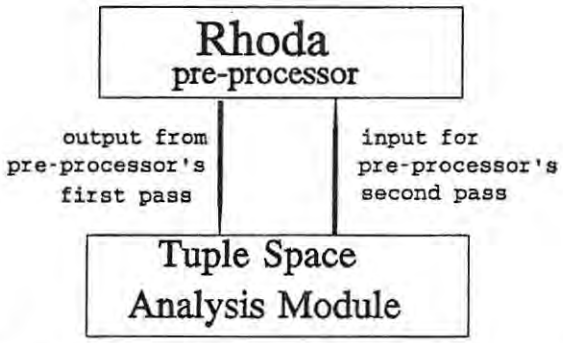


Figure 8. TS analysis module and Rhoda pre-processor interface

6.1.1 Input data

Figure 8 presents a syntax diagram which defines the format and information needed by the analysis module. Each Rhoda operation in the application program generates one input record to this module. The records contain information about the operation line number, the Rhoda operator, the type signature and field parameters of the template, literal values in the template and the Rhoda operation in the source code.

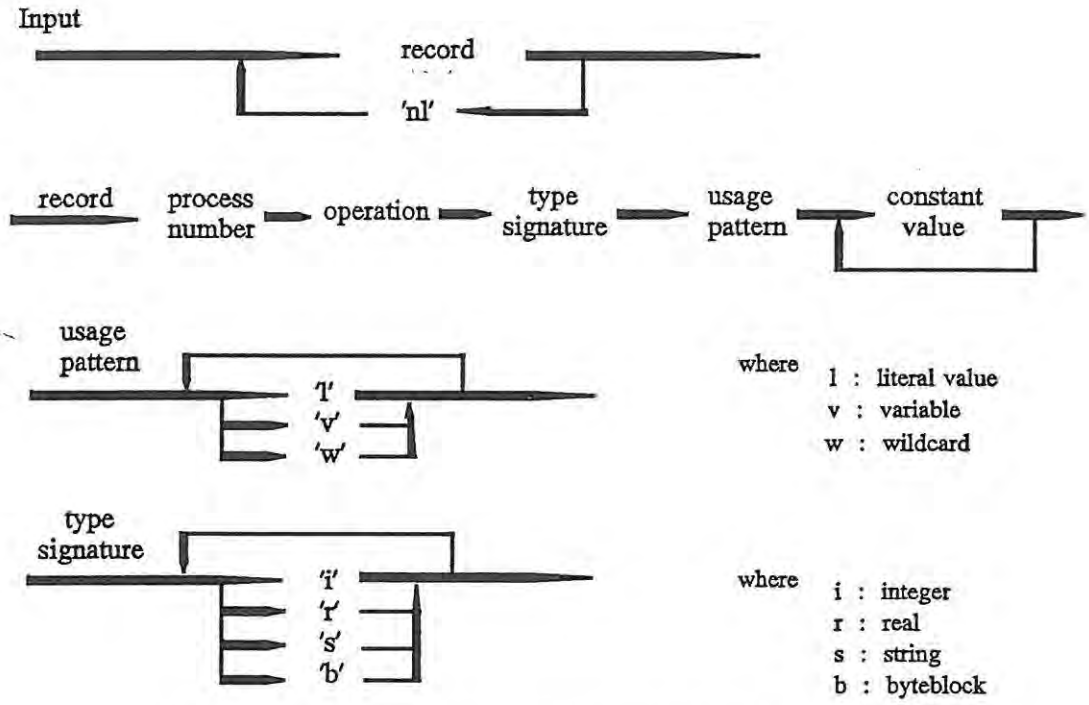


Figure 9. Some syntax diagrams of the analysis module input file.

Sample input data for the matrix multiplication program (section 2.3.1) follows. This is used to illustrate the partitioning algorithms in section 6.2.2 and 6.2.3.

```
44 0 o sib lvv  rows  out("rows", i, p:count)
59 0 i sib lww  results  in("results", ?i, ?R3:num_bytes)
81 0 o si lv  size  out("size", sz)
85 0 o sb lv  matrix  out("matrix", A:sz*sz*sizeof(double))
88 0 o s l mat_wrkr  eval(mat_wrkr)
118 1 i si lw  size  in("size", ?sz)
121 1 r sb lw  matrix  rd("matrix", ?R2:dummy)
136 1 i sib lww  rows  in("rows", ?row, ?R1:bytesz)
144 1 o sib lvv  results  out("results", row, R3:bytesz)
-1 0 I s l mat_wrkr INT_EVAL
```

6.1.2 Output data

The TS analysis module produces three output files. The first file is used to interface with the pre-processor. The information built for this file consists the number of tuple groups created from the analysis, a table and a list of the Rhoda operations in the application program. The table rows denote the tuple groups created. The first column of this table contains the tuple group numbers, while the second column consisting of sub columns, holds information on how processes use the tuple groups. The third column of the table holds the type signature of the tuple groups after the common literal fields of a tuple group have been removed. The last column holds information on the storage scheme for the tuple groups.

The format of the records in the list of Rhoda operations is as follows: the process performing the operation, tuple group number of the template for the operation and the Rhoda operation. This file is used by the pre-processor to convert the Rhoda ideal syntax into concrete C syntax.

Figure 10 presents some of the syntax diagrams which define the format and information produced in this output file.

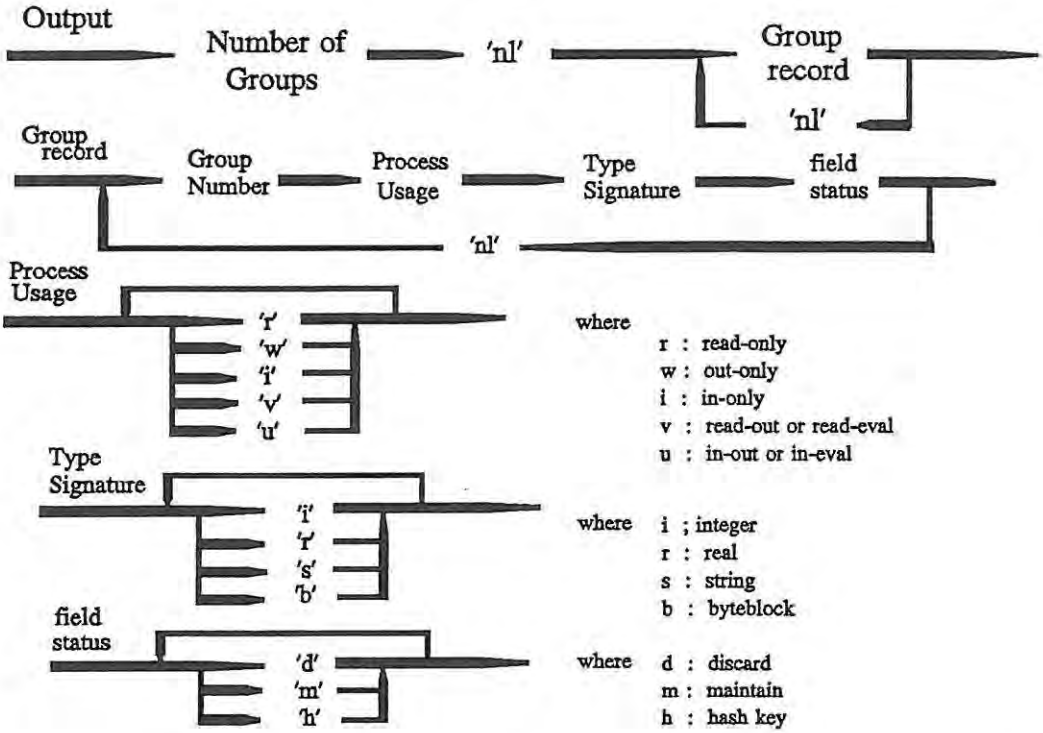


Figure 10. Some syntax diagram of the analysis module output for the pre-processor

Given the sample input data of section 6.1.1, the following is the first output file of the module.

	EXPLANATION
5	number of tuple groups
4 wi - d	
3 wr b dm	table
2 wi i dm	
1 iw ib dmm	(group number, tuple group usage,
0 wi ib dmm	type signature, storage scheme)
1 4 I	
1 1 o	
1 0 i	
1 3 r	
1 2 i	
0 4 o	
0 3 o	
0 2 o	
0 1 i	
0 0 o	Rhoda operations

The second output file consists of a *binary* communication matrix of the placement components of the application program. This file is meant for the cost pre-processor. A cell value of 0 indicate no communication and 1 indicates a communication between the placement components. The dimension of this matrix is $(n+m)x(n+m)$ where n is the number of Rhoda processes and m is the number of tuple groups created. Linda's decoupled nature is illustrated in this matrix. For example, communication between the processes is always zero, and that between tuple groups is also always zero. As an example, the second output file for the input data in section 6.1.1 is given below.

```
                                    EXPLANATION
2                                   number of program components
7                                   number of placement components
0 0 1 1 1 1 1
0 0 1 1 1 1 1
1 1 0 0 0 0 0
1 1 0 0 0 0 0
1 1 0 0 0 0 0
1 1 0 0 0 0 0
1 1 0 0 0 0 0                       binary communication matrix
```

The third file contains debug information which can be used by a programmer in debugging Rhoda application programs. In addition to this file, switches can be set to examine how the analysis module accomplishes its task.

6.2 The Partition Algorithm

Two algorithms have been developed for partitioning TS. We developed an initial algorithm based on the conditions for a tuple - template match to fail discussed in section 5.2. This algorithm agreed in many respects with a similar one in the literature [Zen90] developed at Yale University. But in spite of the tedious nature of the rules, this algorithm does not handle fuzzy cases, for instance case 3 (section 5.2). Consequently we sought a more powerful algorithm with a firm theoretical underpinning. The second algorithm handles all instances of the cases isolated in section 5.2 uniformly. The first algorithm is presented in the section 6.2.2, and the second is introduced in section 6.2.3.

6.2.1 Data Structures

Two linked lists are used in the analysis module. The records of the first list contain information about the Rhoda operations and processes. The second list of tuple groups is created during the partition phase. The records of this list contain the tuple group numbers, the storage strategy for the tuple group, and pointers to individual records in the first list.

The second list presents an efficient mechanism for the provision of information about the tuple groups. This becomes apparent when attempts are made to determine how processes use the tuple groups, to determine

the storage method for the tuple groups, and to produce the output that is used by the Rhoda pre-processor and cost pre-processor.

6.2.2 Initial Algorithm

The information that is needed for the initial algorithm includes template type signatures, their template field parameters and the template literal values. The literal and variable field parameters make up the actual parameters, and the wildcard field parameters are the formal parameters.

The algorithm is as follows:

Step 1: Split the Rhoda operations list into sublists according to their type signatures by building tuple group lists.

Step 2: For each sublists, partition further by common literal values.

Step 2 is carried out as follows:

Let

G_i denote subset i after step 1. and let

$t_k = <f_1, ..., f_N>$ denote tuple k in G_i then

```
for all Gi
 repeat
  for all fj
   if all tk have a literal value in field fj then
    if all fj are the same then
     set fj to be discarded for all tk
    else
     split Gi into further subsets using the fj values
   endif
    else
     set fj to be maintained for all tk
  endif
 endfor
  until no further sub-partitioning for Gi
endfor
```

Using the input data in section 6.1.1, step 1 forms the following subsets of tuples.

Subset 0	Subset 1	Subset 2	Subset 3
0 o sb lv matrix	0 o sib lvv rows	0 o si lv size	0 o s l mat_wkr

1 r sb lw matrix 0	i sib lww results	1 i si lw size	1 I s l mat_wkr
	1 i sib lww rows		
	1 o sib lvv results		

Applying step 2 will set the tuple field flags of subsets 0, 2, and 3 to be discarded or retained. Subset 1 will be partitioned further as follows:

subset 1	subset 4
0 o sib lvv rows	0 i sib lww results
1 i sib lww rows	1 o sib lvv results

This will be followed by setting the tuple field flags of tuple subsets 1 and 4 to be discarded or retained. Although this algorithm handles this example very well, it may not optimise the partitioning for an application program with formal fields in "output" operations.

6.2.3 Second Algorithm

The second algorithm is based on connected graph theory. In this algorithm, we use the rules for a tuple - template match to be successful to link all templates that have common usage. We then extract the connected components which make up the tuple groups. The algorithm is presented below.

1. Use the Rhoda operators to classify the templates into an "*output*" set and "*input*" set.
2. Link "*input*" and "*output*" templates that match to form a bi-partite graph G. (A graph that maps elements of a set P into a set Q, where $P \cap Q = \{\}$.)
3. Report errors by looking for unlinked nodes in G.
4. Find the number of connected components of graph G. This is the number of tuple groups. Get the nodes of each connected component. These are the members of the tuple groups.

Step 1 is carried out by two lists of pointers, one for the "input" set and the other for the "output" sets.

Step 2 linking the templates is accomplished by setting up an adjacency matrix A[][] for graph G. This is carried out as follows:

let u_i denote an "output" templates and
$u_i = <f_1 \ldots f_n>$ where the f_k are the fields.
let v_j denote an "input" templates and
$v_j = <h_1 \ldots h_m>$ where the h_k are the fields.

```
for all u_i
 for all v_j
  A[i][j] = true
  if (type signatures u_i and v_j are equal)  then
   while (exists f_k) AND A[i][j]
```

```
            if f_k and h_k are NOT((both literal and equal) or (actual and
                                formal) or (formal and actual))    then
              A[i][j] = false
            endif
          endwhile
        else
          A[i][j] = false
        endif
      endfor
    endfor
```

In step 3 we check the rows or columns of A for those rows or columns with only zero elements and report the appropriate errors for the operations denoted by such rows or columns.

Step 4 calculates matrix A's transitive closure A+ [Ber73, Deo74, Gri76, Hor77]. The adjacency matrix A shows which "input"/"output" operations interact directly. This demands that they are *"equivalence related"*, in the sense that they must operate on the same tuple group. The transitive closure operation uses the direct relationships to determine those operations that are directly or indirectly related (i.e. they belong to the same equivalence class, or tuple group). A+ has been calculated using Warshall's algorithm [Gri76], which completely eliminates the boolean matrix multiplications and additions involved. Finding the number of connected components of G demands finding the A+'s reduced matrix of A' [Hor77], but calculating the reduced matrix makes us lose the elements of the components of G. The reduced matrix replaces all the nodes in each equivalence class by a single "group" node. We therefore only fold identical rows and not the columns as well. The number of resulting rows then denote number of tuple groups, and the column values of 1 then constitute the members of that tuple group.

This second algorithm is of order $p*q$, where p is the number "input" templates and q the number of "output" templates. The number of elements in both sets $(p+q)$ is usually very small for application programs.

The sample input data given in section 6.1.1. is used to illustrate the second algorithm below.
Step 1 partitions the templates into the following sets:

"output"	"input"
t_0	t_1
t_2	t_5
t_3	t_6
t_4	t_7
t_8	t_9

Applying step 2 forms the following bi-partite graph.

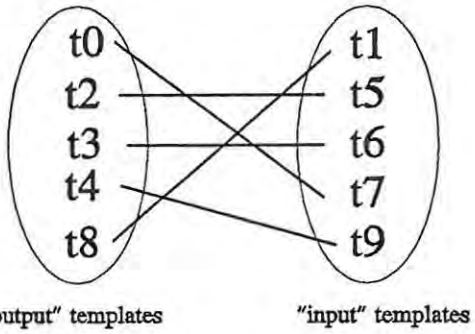


Figure 11. Bi-partite graph of the input data given in section 6.1.1.

$$\text{Adjacency matrix } A = \begin{bmatrix} 0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1 \\ 0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0 \end{bmatrix}$$

Step 3 does not detect any errors.

$$\text{Step 4 gives } A+ = \begin{bmatrix} 1&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&1&0&0&0 \\ 0&0&0&0&1&0&0&0&0&1 \\ 0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&1&0&0&0 \\ 1&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&1&0 \\ 0&0&0&0&1&0&0&0&0&1 \end{bmatrix}$$

Collapsing identical rows in turn gives the following matrix

$$\begin{bmatrix} 1&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&1&0&0&0 \\ 0&0&0&0&1&0&0&0&0&1 \end{bmatrix}$$

Which gives 5 tuple groups, $\{t_0, t_7\}$, $\{t_1, t_8\}$, $\{t_2, t_5\}$, $\{t_3, t_6\}$, and $\{t_4, t_9\}$.

6.2.4 Comparing the algorithms

Although the first algorithm is simpler than the second and it removes redundant fields while partitioning, it is quite ad-hoc and does not detect all cases. The second algorithm has the following advantages:

- it is more elegant.
- it handles all situations uniformly.
- it has a firm theoretical foundation.
- it uses off-the-shelf algorithms.

In view of these, the second algorithm has been chosen in preference to the first in implementing the Rhoda system. The second algorithm is therefore used in discussing the matching analysis below.

6.3 Matching Analysis

After the tuples have been partitioned into tuple groups, two linked lists result which are partially processed. The redundant fields of the templates must be removed, the tuple groups usage by the processes must be set, and the storage schemes for the tuple groups must be removed.

The algorithm used to do these is as follows:

let a tuple group template be denoted as $t_j = <f_1 \ldots f_n>$ where
f_i are the template fields then

```
for all tuple groups
 for all processes
  determine usage
 endfor
 for all j
  a_literal = a_hashkey = true
  while exists t_j AND a_hashkey
   if t_j is not literal
    a_literal = false
   elseif t_j is formal
    a_hashkey = false
   endif
  endwhile
  if a_literal  then
   f_i = discard
  elseif a_hashkey  then
```

```
      f_i = hash key
    else
      maintain
    endif
  endfor
  for all output templates
    if all fields are formal
      discard all fields for tuple group
    endif
  endfor
endfor
```

The usage determined for the program components are in-out, in-eval, read-out, read-eval, in-only, read-only, out-only or eval-only. A process' usage like in-read is treated as in, as rd behaves just like in. To check for tuple group pattern *c* in section 5.4, we check for a group with all "output" templates fields formal, as "input" template of that tuple group will have all fields actual at this stage.

6.4 Summary

This chapter has described the TS analysis module beginning with the file input/output interface with the pre-processor. This interface makes the module language independent and can hence be used in any Linda pre-processor. Two partition algorithms have been described. The second has been implemented in the Rhoda system because it has a firm theoretical underpinning, it is elegant, and it detects all the tricky cases uniformly. The next chapters look at the placement problem.

Chapter 7
Process Placement

In distributed memory systems, process placement (allocation) is an important design criterion. It influences a system's response time and throughput [Chu87a, Han89, Efe82]. In real time applications, where many processes must finish within specified time intervals for the system to perform properly, the importance of process placement cannot be overlooked. In parallel machines, the principal objective of process placement is to improve the *speedup*[1] and hence the *efficiency*[2].

It would not be unreasonable to expect the speed of a distributed memory system with n processors, each having a processor speed of k, to be $n*k$. Unfortunately this is not the case - the speed attained is less. The difference is caused by factors such as control overheads, inter-processor communication, unbalanced processor loading, queueing delays, and **precedence relationships** (precedence order of the program parts assigned to separate processors). The goal of a placement algorithm is to minimise one or more of the above factors so as to improve the performance of the system.

7.1 Parallel Algorithms

A placement algorithm for Rhoda should be general enough to cater for all, or at least, most classes of Rhoda programs. The literature was surveyed for classes of parallel algorithms so as to characterize Rhoda programs. The taxonomy encountered for parallel algorithms varied from one source to another. [Fin88] comments that finding a suitable taxonomy for all parallel algorithms may be too ambitious an undertaking, as there is the need for a better understanding of where parallelism can be found and how it can be expressed in programs and algorithms. Some taxonomies encountered are surveyed below.

[Car89b] discusses programming techniques and the transformation of parallel algorithms of one class to another. He illustrates his examples with Linda and distinguishes three classes of parallelism. These are:

- result parallelism: each process produces one piece of the result.
- agenda parallelism: an agenda of activities is specified and a process is assigned to help with the current item on the agenda, for example, the master-worker methodology (farm of workers) falls into this category.
- specialist parallelism: each process performs a specific job and is connected into a logical network, for example, pipeline and systolic computations fall into this class. A systolic

---

1. Speedup: Time taken to execute the best sequential algorithm solving problem A divided by the time taken to execute a parallel algorithm solving problem A.

2. Efficiency: Ratio of Speedup to number of processors.

computation is a pipeline algorithm which performs identical computations at each segment, and has a rhythmic and regular flow of data in more than one direction.

[Qui88] considers the design of efficient parallel algorithms for MIMD computers. He isolates three classes of parallel algorithms:

- pipeline algorithms: pipeline or systolic computations (similar to [Car89b]'s specialist parallelism).
- partition algorithms: programs are divided into sub problems that are solved by individual processors and then combined to form the solution. In this class, the processors synchronize their activities. These are also referred to as synchronized algorithms.
- relaxed algorithms: These algorithms work without process synchronization (asynchronous algorithms).

[Nel88] classifies programming paradigms for Nonshared Memory Parallel Computers as follows:

- compute-aggregate-broadcast (CAB) algorithms: algorithms which have three phases: a compute phase which performs some basic computation, an aggregate phase which is usually a tree-based computation that combines data from processes into a single global value, and a broadcast phase which returns the global information or a directive based upon it to all processes. An example is the Jacobi iterative method for solving Laplace's equation on a square [ibid].
- pipeline algorithms: pipeline or systolic computations.
- divide and conquer algorithms: problems are divided into two or more smaller problems which are solved independently. The results combine to give the final result (an example of [Car89b]'s agenda parallelism).

[Fin88] deliberates on large grain parallelism and distinguishes the following classes of distributed-algorithms:

- generate and solve algorithms: consists of a master process which divides a problem into sub-problems and a pool of slave processes which stands ready to solve these sub-problems. This is an example of the agenda parallelism of [Car89b].
- iterative relaxation algorithms: divides the data space into regions which are parcelled out to different communicating processes.
- passive data pool algorithms: a large data space is managed by many processes (e.g distributed file systems).
- systolic algorithms: pipeline or systolic computations.

The above taxonomies all identify pipeline and systolic algorithms as a class. It is also observed that examples of agenda parallelism fall into other classes in the other taxonomies. The class of partition algorithms [Qui88],

of which message passing systems are a member is confined to [Qui88]. [Car89b] indicates that this class of programs is easily transformed to that of agenda parallelism which is ideal for Linda. Since no strong agreement is evident, and because [Car89b]'s taxonomy was developed while discussing programming in Linda, we have adopted [Car89b]'s classification of Result, Specialist and Agenda parallelism styles for Rhoda programs.

7.2 Linda's programming methodology

Result parallelism naturally suits problems whose goals are to produce a series of values with predictable organization or inter-dependencies. These algorithms are programmed using *live data structures*. A live data structure consists of a number of active processes, each computing one data element of the final structure. The processes already carry information about the organisation of the final result, and as each process completes, it becomes a data value. For example, the multiplication of two matrices may involve spawning *NxM* live tasks, each with its own indices, and each one finally producing one value in the result matrix. This model may produce fine grained programs which will have substantial overheads due to creating and coordinating large numbers of processes [Car89b].

Specialist parallelism is suited for algorithms which are best understood as a network. These are programmed using *message passing*, a model in which all data objects are encapsulated within explicitly communicating processes. For example, in a graphics program on 3-dimensional transformations, a process could be assigned to generate and to display the objects, another to rotate the object, a third to project the 3-dimensional coordinates onto 2-dimensional coordinates, and the fourth to remove hidden surfaces. This paradigm may involve too many processes and too much inter-process communication [ibid].

Agenda parallelism algorithms are formulated as *distributed data structures*, a paradigm in which processes communicate and coordinate by leaving data in shared objects [ibid]. This paradigm suits problems that transform or perform a series of transformations to all elements of some set in parallel.

To build efficient programs, Linda programs are transformed from one type of algorithm into another. Efficient programs are mostly produced by algorithms programmed under the master-worker (*replicated worker* [Ahu88]) model [ibid]. This replicated worker model is the most flexible embodiment of Agenda parallelism in Linda, and is therefore the main programming methodology of Rhoda. An example of this model is the matrix multiplication program presented in section 2.3.1.

Most programming languages do not support this replicated worker model well. There are a number of process placement modelling tools which are used to help understand parallel programs. We consider two of these, and comment on the characteristics of Rhoda programs when viewed in terms of these tools.

The first useful tool is a *control-flow (dependency) graph* [Chu87a, Nan90], which represents the logical structure and precedence dependencies amongst the modules (rectangular blocks) to be placed. A task *A* is said to have precedence over task *B* if it must finish its execution before task *B* can start. If precedence analysis can detect that processes are time disjoint, it is possible to schedule them on the same processor. Precedence analysis can also be useful if processes can be assigned different priorities within a processor.

Linda has no direct support for precedence relationships. As such, we need extra information to cater for precedence relationships or to implement a dynamic response to implement the scheduling of such processes at run-time. However, a Linda program can explicitly finish one section before issuing an *eval* operation. A programmer can "dynamically" schedule processes to cater for precedence relationships. This is the usual style of programming problems with strong module dependencies. We do not cater for precedence relationships and consider it to be beyond the scope of this work.

Conventional network-style parallelism partitions a program into a large number of simultaneous modules synchronizing their activities to derive parallelism *(synchronized algorithms)* [Wan90]. A number of these modules usually have strong precedence dependencies. If replicated worker model programs are formulated such that its processes have weak/minimal dependencies, their processes will have almost smooth continuous loads. Sub-processes of this model are created by duplicating identical copies of a single process as opposed to partitioning. Their weak dependencies makes it feasible to simultaneously start all processes and tuple groups. Their programs therefore have a single parallel AND branching control-flow graphs. Figure 12 below illustrates the control flow graph of the matrix multiplication program presented in section 2.3.1.

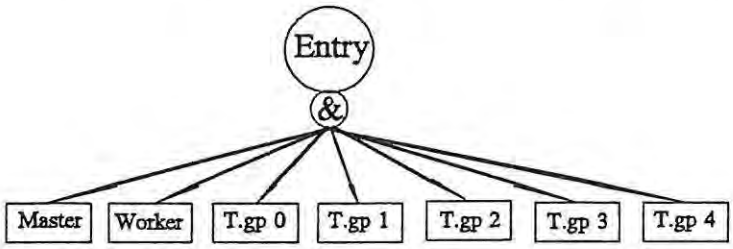


Figure 12. Control-flow graph of matrix multiplication program

Another widely used tool is a process graph. The weighted nodes of this graph represent component loads and the weighted edges represent the communication costs.

The *process graph* of Rhoda programs is usually simple with few nodes as there are few processes. It reveals that processes communicate with only tuple groups, but not with one another. This decoupling must be considered in a clustering algorithm. Figure 13 illustrates the process graph of the matrix multiplication program presented in section 2.3.1. *M* denotes the master process, *g0,..g4* denote the five different tuple partitions, and *W* denotes the worker process. The arcs denote information flow, or coupling between the

components.

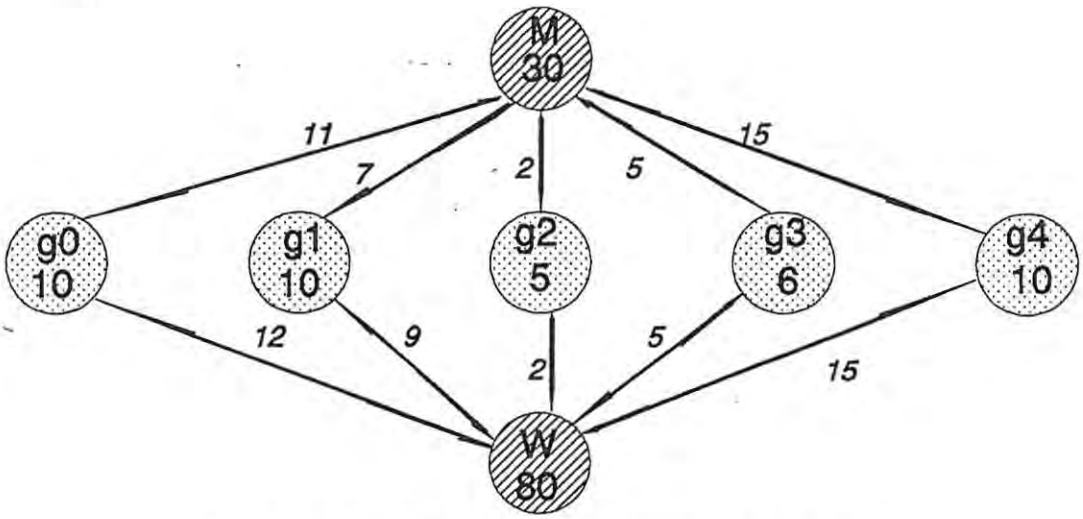


Figure 13. Process graph of matrix multiplication program

The replicated worker method executes in the same way irrespective of the number of worker processes. The worker processes of this model look for tasks that need to be computed. The tasks are therefore distributed at run-time among the available processes [Ahu86], usually one per processor to achieve dynamic load-balancing.

7.3 Requirements of the placement Algorithm

Taking cognisance of the *"replicated worker"* programming methodology, the implementation of a distributed partitioned TS, and the transputer network hardware of Rhoda, we extract the requirements, limitations, and special circumstances of the placement problem. The requirements of the placement algorithm are as follows:

1. The algorithm must cater for agenda, specialist and result parallelism styles.
2. It simplifies the placement algorithm if tuple groups are treated just like other processes. Collectively, the data groups and processing elements are called components.
3. The algorithm must cater for replicating processes, but no process and its replicate should reside on the same processor so as to avoid pointless context switching.
4. No tuple group must be replicated.
5. The algorithm must cater for p components controlling peripheral hardware where $p \leq$ the number of transputers in the network.
6. Each node in the network must be capable of running a TS server so as to implement a distributed TS system.

The limitations of the algorithm are as follows:

1. Apart from processors that control peripheral hardware (e.g. IO), a homogeneous network

is assumed for the rest of the system.

2. Connections between processors are fixed beforehand and all have identical capacities. The algorithm must be able to work for arbitrary topologies, but it does not suggest or prescribe a topology.
3. A component may use only one peripheral device as each node may control only one peripheral device.
4. The number of replications for each component is given to the algorithm, and not suggested by the algorithm.

The special circumstances are:

1. We assume that Rhoda application programs solve coarse grain problems (that is chunks of computing processes with little time for communication).
2. Each component has a processing cost and processor load estimate supplied beforehand.
3. We ignore precedence dependence and assume the Rhoda programs are formulated or used in situations with minimal precedence dependencies.
4. In view of the automatic dynamic load balancing which is inherent in the replicated worker model, an initial placement allocation problem is to be solved.

Rhoda program components could have strong precedence dependencies. Placing Rhoda programs with strong precedence dependencies falls outside the scope of this thesis, and is therefore not considered.

Although the worker processes of the replicated worker model dynamically balance the work load amongst themselves, network traffic and processor overloading due to tuple group servers are potential bottlenecks. Processes and tuple groups must therefore be allocated to the processors with a view to reducing these bottlenecks.

7.4 Methods of Process placement.

Process placement can be either dynamic or static. In dynamic allocation, components are allocated to processors at run-time. Dynamic allocation has the advantage of being responsive to the run-time state of the processors [Hil88]. Disadvantages include increased processing time to decide on and implement the altered allocation. A survey of a number of dynamic allocation algorithms are given in [Gaj85] and [Mac89].

Static allocation maps components onto processors before the execution of a program. This allocation has the advantage of paying allocation costs only once for a program's execution. But this requires compile-time predictions of how the program will behave at run-time. Hence the allocation is based on guesstimates.

Static allocation of components was chosen, and three allocation models were distinguished: *integer*

programming, graph theory methods, and *heuristic methods*.

The integer programming method is based on an enumeration algorithm. [Map82] gives a description of this placement model. It has the advantage of producing an optimal solution to the optimization problem. However, this allocation method has been found to be slow [Kas84, Han89]. Further, the placement optimization model is only a crude approximation of the real world. Finding an optimal solution using guesstimates is therefore not useful. In the light of these two disadvantages, an integer programming method has not been used for the Rhoda placement algorithm.

The processes to be allocated using graph theoretic methods are represented as a set of nodes connected by weighted non-directed arcs which represent communication costs between the nodes. This allocation method minimizes total Inter-processor communication cost and the total processing cost by performing a max-flow/min-cut [Tan81] on the graph. This method is faster than integer programming but becomes very complex and computationally expensive for a network of processors in excess of two [Han89].

Heuristic algorithms form logical clusters to balance processor loads while minimizing inter-cluster communication costs. These clusters are formed using the communication costs and processing costs subject to constraints such as processor memory and real-time considerations. Unlike the integer programming model, heuristic methods aim at finding sub-optimal assignment [Han89, Efe82, Chu87a]. Heuristic methods have been found to be faster, more extensible, simpler, and in some cases the only technique that can be used to solve difficult placement problems [Efe82]. From these advantages, a heuristic method has been employed for the Rhoda placement algorithm.

7.5 Comparison of some heuristic algorithms

A number of heuristic algorithms were considered. The overview of a few that were found to fulfil partially the requirements, limitations and special circumstances of the Rhoda placement are presented below.

The simple classification algorithm by [Gyl76], searches for pairs of modules such that when these are assigned to the same processor (fused), the greatest inter-process communication cost is eliminated. This algorithm is simple but does not provide a mechanism to guarantee that the number of clusters found will not be more than the number of available processors. This algorithm does not cater for module replication, and was deemed unsuitable.

A second algorithm [ibid] defines *initial centroids*[1] for each candidate cluster. Using these centroids the

[1]. centroid here refers to the mean communication volume for the cluster.

algorithm forms a fixed number of clusters to ensure the clusters are not more than the number of processors. But finding eligible pairs of processes and/or clusters is very expensive [Chu87a]. This algorithm, like the first, was deemed not suitable because it does not cater for module replication, and in the worse case it may not converge.

Another algorithm, *Module Clustering Algorithm* [Efe82] forms clusters to obtain the minimum inter-processor cost without considering any constraints. If the clusters meet the load balancing constraints, the algorithm terminates, otherwise a *Module Reassignment Algorithm* shifts modules from the overloaded processors to under-loaded processors. In the worst case, this algorithm does not converge [Chu87a], neither does it cater for module replication.

The *Task Response Time Model* and the *Module Assignment Algorithm with Module Replications* [Chu87a], were designed primarily to cater for programs with strong precedence dependencies between their modules. Although the algorithm caters for module replication, it was designed for a real time system and is therefore heavily oriented towards satisfying minimal time constraints. This makes it too complex and unsuitable.

These algorithms and others in the literature [Mck88, Qui84, Sad90, Sch89, Shi89, Siv89, Wan90 and Nan90], all assume there are more components to be placed than processors. But this is not usually the case with Linda programs. There are usually more processors than processes. It is therefore imperative that worker processes are replicated in order to maximize processor utilization. From the difference in characteristics between existing placement algorithms and the Rhoda placement problem, we observe that the Rhoda placement problem is distinct and therefore needs its own placement algorithm.

7.6 Summary

Classification schemes for parallel algorithms have been surveyed in this chapter and one which occurs in the literature in conjunction with Linda programs has been adopted for Rhoda. The chapter also extracts the requirements, restrictions and limitations of the Rhoda placement algorithm. Placement methods are surveyed in relation to Rhoda. A static heuristic approach is selected for Rhoda's placement problem. An overview of a number of heuristic allocation models is also scrutinized in an attempt to determine their suitability for the Rhoda placement problem. Due to the unsuitability of existing algorithms, the next chapter develops a new algorithm for Rhoda.

Chapter 8
The Rhoda Components Placement

The factors that contribute to the run-time overhead of the Rhoda system include the Helios [Hel89] support for TS servers (appendix 1), and the queuing delays within TS due to a process acquiring exclusive access for update operations. Process allocation cannot directly influence these delays. However, the communication overheads and the processor loads which also contribute to the run-time overhead can be minimized by judicious component placement.

A *Resource Map* file within the Helios operating system defines the configuration of the network. The details of this file include the transputer interconnections and attributes for each transputer node. This file can be used to obtain the necessary information about the network topology for the placement algorithm. Helios dynamically establishes paths between any two transputer nodes in the network. In the Helios-based Rhoda, a component can be executed on any node in the network without the need to reconfigure the program.

8.1 An initial Process Placement

A fast heuristic algorithm has been developed to minimise **Inter-processor communication (IPC)** cost while balancing processor loads at compile-time. This initial algorithm serves as the basis of a dynamic allocation scheme currently under investigation at Rhodes University.

The allocation algorithm is based on cluster analysis. It logically classifies the Rhoda components into clusters while satisfying load constraints such that the inter-cluster communication costs are minimized. The clusters are formed such that each one will have its own node on which to execute. The logical clusters are then mapped onto the transputer nodes such that the total network traffic is minimised.

The use of the term clustering here is different from its traditional use in multivariate analysis. In multivariate analysis, objects are described by a set of numerical measures. Clustering refers to devising a classification scheme for grouping the objects into a number of classes such that groups are similar in some respect and unlike those from other classes [Eve80]. Clustering as used within this work refers to grouping components with predefined costs into identical groups such that communication between them is minimised.

8.1.1 The Allocation Environment

Two files describe the allocation environment for the placement algorithm. These files are the configuration file (Hardware information) and the application description file (information on the components). From the Helios Resource Map (configuration file), we obtain the following physical parameters of the transputer network:

n the number of transputers in the network.

Td(n,n) the connectivity matrix of the transputers in the network. The matrix element (Td[i][j]) has a value of 1 if the nodes *i* and *j* are connected directly. The other element values denote the shortest distance between nodes with no direct links. This distance is calculated in terms of the number of intermediary hardware links.

N(n,p) peripheral connectivity matrix for the transputer nodes (p = number of peripheral controllers in the network). This matrix describes the nodes in the network that control peripheral devices (eg I/O).

From the Rhoda cost pre-processor (compiler/pre-processor), the following physical parameters are obtained prior to allocation:

M(n) maximum load vector for the transputers in the network.

I(n) Initial load vector for the transputers in the network.

From the TS analysis module and the Rhoda cost pre-processor (application description file), the following parameters, vectors and matrix are obtained prior to allocation:

m the number of components to be placed in the application program (graph nodes). This is extracted by the TS analysis module.

C(m,m) matrix of data transmission costs between the components to be placed. Each cell C[i][j] denotes the communication cost between components i and j.

E(m) the component load vector. This vector gives the processing cost of each Rhoda process and tuple group.

R(m) the component replication vector. A vector element R[i] denotes the number of instances of component *i*.

H(m) components peripheral hardware requirements vector. This vector gives the peripheral hardware (if any) required by the components.

8.2 The Allocation Problem

The Rhoda allocation problem has been formulated as an optimisation problem over the space defined by the transputer network and the components to be placed.

Let v be number of clusters formed. where $v \leq n$,

x_{ij} denote inter-cluster communication between ith and jth cluster.

t_i denote the possible maximum load permitted for each cluster.

O_i denote the load of cluster i.

and let

$$Q = \sum_{i=1}^{v-1} \sum_{j=i+1}^{v} x_{ij}$$

= total inter-cluster communication cost.

An ideal solution to the placement problem requires that conflicting requirements be satisfied. On the one hand we would like to have an optimally balanced processor load. But this might only be achievable if we are prepared to move large traffic volumes in the network. On the other hand, clustering components to reduce the traffic volumes might unbalance the processor loads. Some compromise is necessary, and this is achieved by requiring approximate rather than exactly balanced processor loads.

Let

α be the tolerance accepted for a balanced cluster load. Usual values are between 10% and 20%.

The objective of the algorithm is then to find the allocation vector P(v) such that Q is minimised subject to the following constraints

$\forall i \qquad t_i - \alpha t_i \leq O_i \leq t_i + \alpha t_i$ and that

- no component is replicated within the same cluster.
- and no tuple group is replicated within the system.

Placing two identical Rhoda processes on a processor could improve performance of the system, if tuples were always readily available, but their matching overhead was quite heavy. One process could get scheduled while the other was busy waiting for input from TS. This constitutes a form of "double buffering". However, this optimization falls outside the scope of this thesis, and is not considered.

8.3 The Allocation Algorithm

The placement algorithm is as follows:

1. From the application description file, read the relevant data of the components and the processor loads.
2. From the Helios resource map file, extract the hardware map of the network.
3. Verify that the required replication in the vector R can be accommodated by the network while satisfying the replication constraints (i.e. no component and its replicate will be assigned to the same cluster and no tuple group is replicated). If the required replication cannot be accommodated by the network, stop.
4. Modify the communication cost matrix C, components load vector E, and the peripheral hardware requirements vector H to cater for the replicated components.
5. Cluster the components.
6. Balance the cluster loads to satisfy the tolerance criteria.
7. Calculate the inter-cluster communication costs for the clusters.
8. Allocate clusters that need certain fixed processors using the peripheral connectivity matrix

N(n,p), and the components peripheral requirements vector H.

9. Assign the remaining clusters using the transputer connectivity matrix Td such that the total IPC cost is minimised.

It is necessary for the algorithm to check if the network can accommodate the required replications, since the vector R is supplied to the algorithm. To do this the algorithm first forms clusters without replications in step 3. An examination of the placement components in each cluster, their processing costs, the cluster loads, and the processor loads makes it possible to decide if the required replication is feasible. The replication is deemed not feasible if any one of the following is true:

- the total load of the placement components exceeds the capacity of the network.
- if the replication will cause a component and its replicate to be assigned to the same cluster.
- a tuple group is replicated.

8.4 Clustering Rhoda components

Figure 14 is the process graph of the matrix multiplication application program (section 2.3.1) after replication. The single worker shown in figure 13 has now been replicated three times, and the traffic loads have been re-distributed. The processing costs of the master process and tuple groups remains the same as they are not replicated. Because the worker processes continuously look for tasks at run-time just like the single worker process, each worker process is assigned the processing cost of the single worker. Similarly the communication cost of process *B* with the tuple groups remain the same.

Re-distributing the traffic between tuple groups and the replicated processes entails determining whether a tuple group will share its traffic between the replicated workers or will increase the network traffic as the number of workers increases. The latter occurs when each replicated process needs its own copy of global data. Under such circumstances, even if the network has a fantastic connectivity and many processors, this increasing traffic cost may outweigh the advantages of further replication. Determining such circumstances falls outside the scope of this work and is therefore not considered. In figure 14, *g0*, *g1*, and *g4* share their traffic from figure 13, while the global traffic from *g2* and *g3* must be transmitted to each new worker.

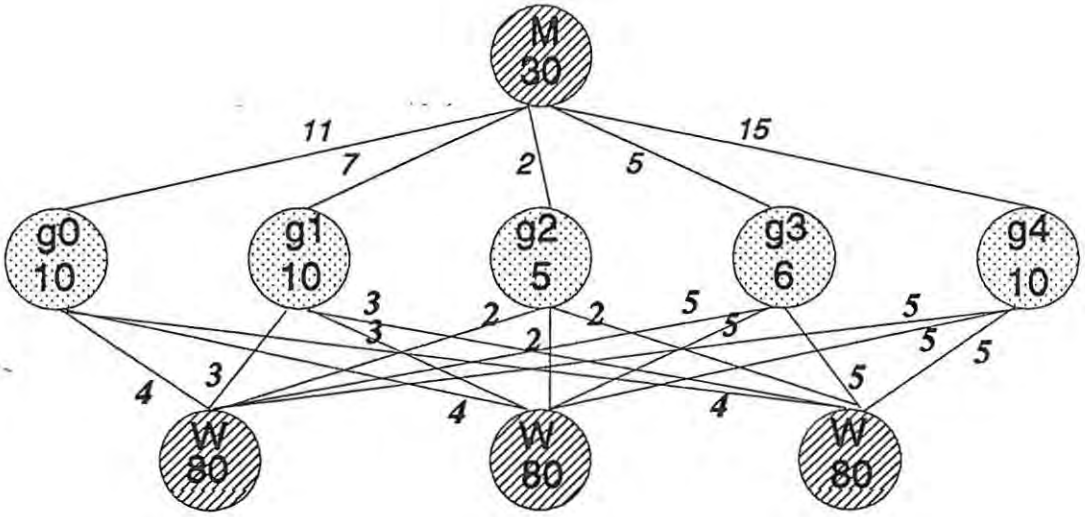


Figure 14. Matrix multiplication process graph after replication

Inspection of the process graph reveals that a tuple group communicates with every process. This suggests that a tuple group be clustered with the process having the heaviest communication.

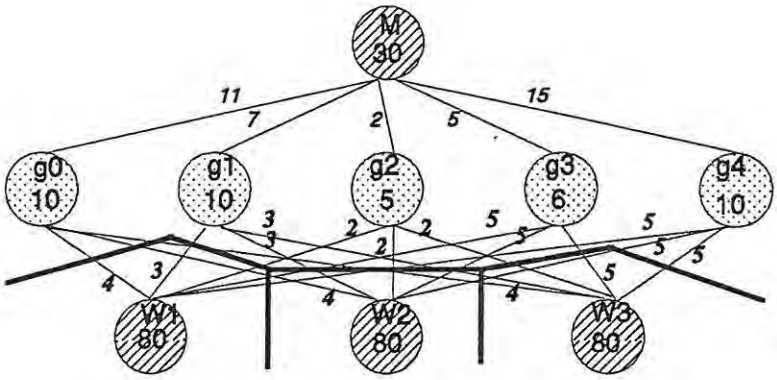
We assume the processing costs of Rhoda processes to be heavy and that of tuple groups are relatively light. A process can therefore be clustered with a number of tuple groups. Communication costs are expected to be smaller than the processing costs of processes, but it is the dominant cost that must be minimised. The clustering algorithm therefore aims to minimise the inter-cluster communication costs. The algorithm starts with a cluster and assigns components until the cluster is full or no suitable component can be found. The algorithm then proceeds to create new clusters until no more clusters can be created. The main steps in the algorithm are as follows.

1. Initialise the cluster loads.
2. Select an attached component (a component that requires a peripheral hardware) using vector H as a *centroid* (component to be used to select tuple groups for this cluster). If there is no attached component, then select a component.
3. Update the current cluster load.
4. Select an unassigned tuple group that has not been considered for this centroid with the highest (next highest) communicating cost. The communicating cost becomes a candidate *inter-module communicating cost (IMC)*. IMC is the communication cost between modules resident on the same processor. If all tuple groups have already been considered, start a new cluster by repeating step 2.
5. check if the combined load of the selected tuple group and the current cluster satisfy the tolerance constraint. If not, select the next tuple group by repeating step 4.
6. get the highest communicating cost *(cut-cost)* of this tuple group with the other components

not clustered. Compare the cut-cost with IMC. If IMC is less than the cut-cost, discard this candidate and select the next tuple group by repeating step 4.

7. cluster the tuple group and update the current cluster load. Repeat step 4 until no more tuple groups can be located for the current cluster.
8. repeat steps 2 through 7 until all the components have been clustered.

The process graph and table in figure 15 illustrates how the components in figure 14 are clustered.



Cluster Number	Component	CI Load	IMC	Cut Cost	Cluster ?
1	M	20+30			Yes
	g4	60	15	5	Yes
	g0	70	11	4	Yes
	g1	80	7	3	Yes
	g3	86	5	5	Yes
	g2	91	2	2	Yes
2	W1	80			Yes
3	W2	80			Yes
4	W3	80			Yes

Figure 15. Clustering the components of the matrix multiplication program

In the application above, process *M* needs an IO device. The node controlling the IO device has an initial load 20 and a maximum load of 110. The other nodes in the network all have an initial load 0, and a maximum load of 100.

Tuple groups which are not clustered form clusters of their own. These tuple groups are assigned to the main clusters when cluster loads are balanced. The algorithm is therefore guaranteed to converge.

8.5 The required Cost functions

The various costs (guesstimates) needed by the Rhoda placement algorithm are:

- the communication costs between the components to be placed.

- the processing costs for the processes and the tuple groups.
- and the initial and maximum loads for the transputer nodes in the network.

At this stage, these cost estimates must be supplied manually. The literature does not give much guidance on automatic estimates of these costs. The implementation of this aspect is regarded as being beyond the scope of this research. However, we do overview some suggestions on estimating these costs in chapter 9.

8.6 Summary

We have presented a new placement algorithm for the Rhoda system in this chapter. This heuristic algorithm allocates components to nodes in a transputer network with arbitrary topology. The allocation is carried out in two phases: logical clustering of the components and placing the clusters on the transputer nodes. The algorithm operates under the following constraints:

- the processes have no (or minimal) precedence dependencies.
- the programs are solving coarse grained problems.
- it does not suggest a topology.
- it assumes each node in the network is capable of running a TS sever.
- it assumes all cost functions are supplied beforehand.

The estimates required by the algorithm are :

- the computational capacity of each node in the network.
- the processing loads of the processes, and the tuple groups.
- the communication costs between the processes and tuple groups.

Chapter 9
Evaluation of the TS Analysis and Placement Modules

This work is evaluated in relation to the Rhoda system of which it forms an integral part. A qualitative evaluation of the analysis module is presented below. We also present a qualitative evaluation of the placement module, because the algorithm has been developed ahead of the lower levels of TS implementations. Whereas the algorithm is able to place tuple groups on different processors, the tuple space server is currently only able to handle a centralized TS.

9.1 The Analysis Module Performance

The file input/output interface of the TS Analysis module makes it language independent. This module can therefore be interfaced with any language's Linda pre-processor.

The partitioning of TS has made it possible to move the Rhoda system from a centralized TS to a distributed TS implementation. The ability to distribute TS partitions has allowed us to alleviate the concentration of network traffic to one centralized node. The automatic placement achieves this by minimising the network traffic.

Partitioning TS makes it possible for the Rhoda pre-processor to transform the ideal syntax of Rhoda programs into concrete C syntax which directs each Rhoda operation to the appropriate subset of tuples. With "input" templates aimed at a tuple group, this reduces the run-time matching overhead. The partition algorithm has a solid graph theoretic foundation and gives an optimal TS partitioning.

The matching analysis identifies five run-time matching cases in order to further reduce the tuple retrieval overhead. This analysis reduces the run-time matching of two cases to semaphore operations (section 5.4). A FIFO queue is also used to eliminate run-time matching in a third case, and to effect transfer of data from actual to formal fields. The hash table used in the fourth case provide keys which selects tuples optimally within a group.

Removing redundant tuple fields (section 5.3.1) reduces the transport buffer sizes, and hence the length of tuples transferred between the components of an application program. This results in short message transfers within a network. The usage analysis also makes it possible for the Rhoda pre-processor to prevent processes from accessing tuple groups they do not use.

From the evaluation above, we deduce that tuple retrieval which is a potential bottleneck for Linda systems [Dav89], is redressed by the TS Analysis module. Although the TS Analysis module adds to Rhoda's compile-

time, the algorithm is fast and forms a small percentage of the compile-time. The gains of this module are essential for a good Linda implementation.

9.2 The placement Module Performance

The goal of the placement algorithm, is to replace the existing manual algorithm with an automatic technique at compile-time. We therefore compare the algorithm placement with a hand placement. Five test programs have been selected from the three classes of Linda programs which are identified in chapter 7.

The first is a result parallelism solution to finding the number of primes between 1 and n [Car89b]. Although this solution is not efficient [ibid], the interest here is in the placement. There is only one tuple group with n elements (active tuples) forming the result vector. We expect all the active tuples to be placed on a single processor, and the workers to be placed on the other processors within the network. Figure 16 illustrates the process graph of this program and the algorithm's placement (for n = 5) on the transputer topology in figure 17. It is observed that the clusters have been allocated to minimise the network traffic. This placement is similar to the expected manual placement.

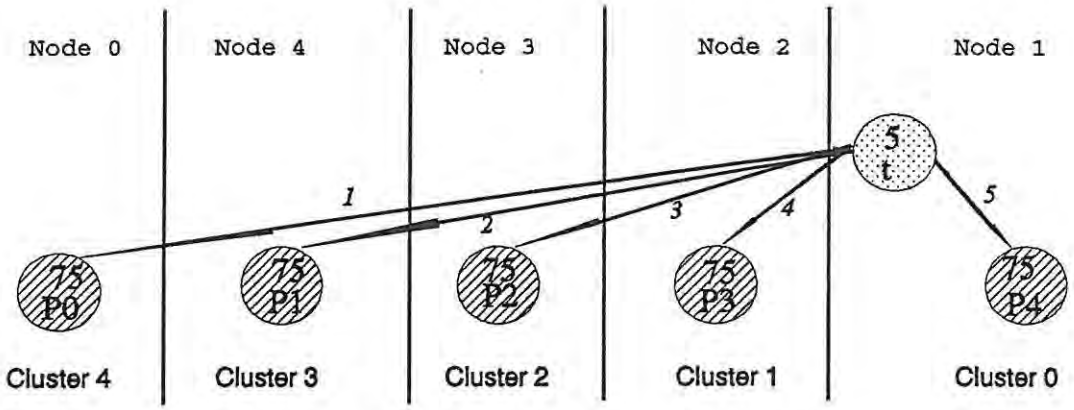


Figure 16. Process graph of a result parallelism solution for finding primes

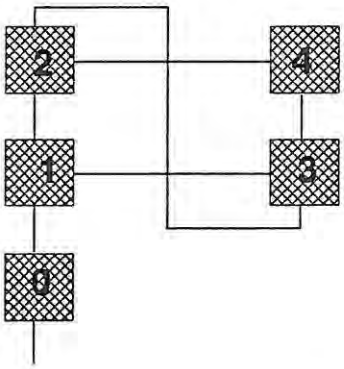


Figure 17. Transputer topology for figure 16

Three agenda parallelism (replicated worker model) programs were placed by the algorithm and by hand. One of these programs is the matrix multiplication program that was used to illustrate clustering (section 8.4). Figure 18 illustrates how the clusters formed in figure 15 were placed onto the transputer topology. The algorithm's placement for this program, a 12-queens program [Cla90] and a ray-tracing programs [ibid] are similar to the hand placed ones. The hand placement of these programs on 1, 2,...,8 transputers yields almost linear speedups [Cla90]. The placement algorithm therefore does not degrade the run-time of the Rhoda system.

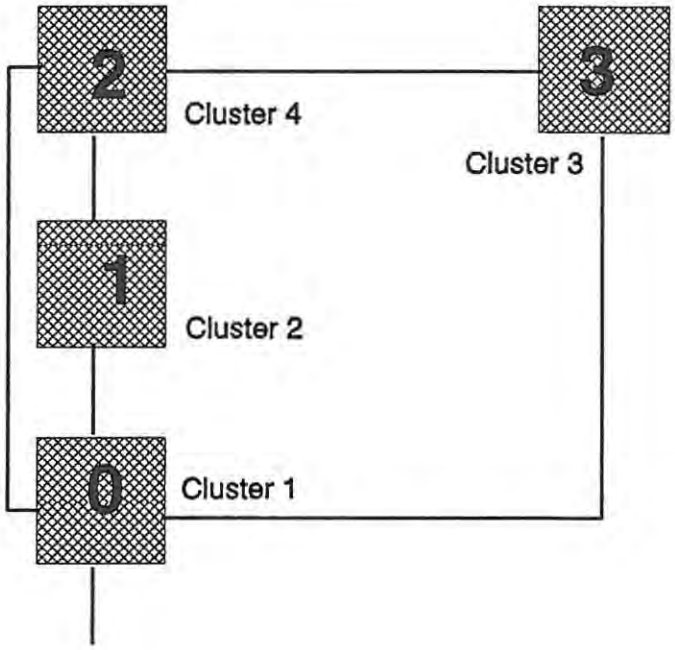


Figure 18. Transputer topology and placement of the clusters formed in figure 15 (matrix multiplication program)

The fifth is a specialist parallelism program performing 3-dimensional transformations (section 7.2). TS analysis partitioned TS into eight partitions. When hand placed, the four processes and the tuple space partitions must form a pipeline. The algorithm also placed the eight tuple groups and processes in the form of a pipeline network. The process graph in figure 19 illustrates the algorithm placement on the topology given in figure 20.

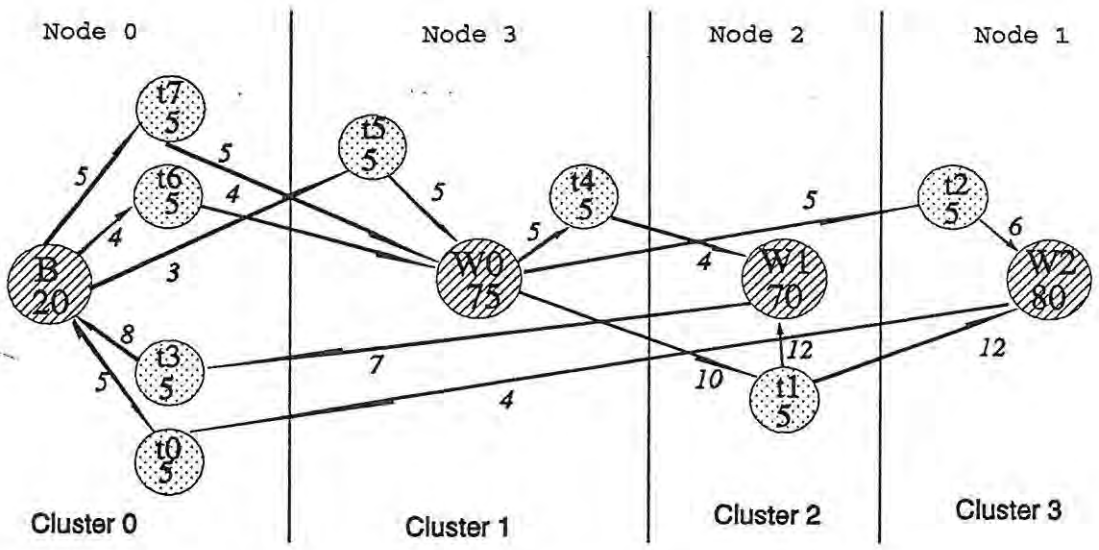


Figure 19. Process graph of a specialist parallelism program

Process *B* creates objects which are placed in *t5*. *W0* does figure rotation and places its results in *t4*. *W1* does projection placing the results in *t3*, and *W2* does the removal of hidden surfaces placing results in *t0*.

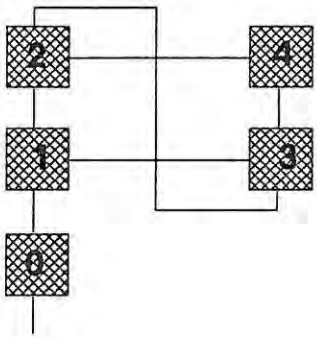


Figure 20. Transputer topology for figure 19

We observed the algorithm to perform as well as manual placements. In complicated programs (for example figure 19) where there are many variables to juggle, we expect our algorithm to simplify the task and to outperform the programmer. The algorithm is also fast and does not degrade the run-time of the Rhoda system.

However, ignoring precedence relationships between processes (for example figure 16) can degrade the algorithm for processes which have strong precedence. In this case the static analysis may not match the dynamic case, as the processes may not be inherently parallel. But this is a weakness of Linda which can be tackled by transforming the program to an efficient version [Car89b] or catered for by a dynamic allocation scheme.

Chapter 10
Future Work

Linda systems for other base languages can be developed with the aid of the tuple space analysis module. This module, which can interface with any base language's Linda pre-processor will optimise the retrieval of tuples for a good implementation. A heterogenous Linda system involving a number of base languages can also be developed to support the claim that Linda can coexist peacefully with a number of base languages [Gel85].

The literature does not give much guidance on estimating processing costs and communication costs at compile-time. This is an area that is crucial for automating the allocation of processes and tuple groups to processors at compile-time. [Vra89] suggests lines of machine language instructions as an estimate of processing costs and [Chu80] suggests using branching probability and loop frequencies. An investigation into the behaviour of Rhoda programs is therefore necessary to perform more compiler/pre-processor analysis to provide these costs.

As Linda programs are usually written such that processes are dynamically scheduled, the costs derived above should provide extra information to take into account precedence relationships between processes and on *eval* operations. With this extra information available, it may be possible to extend the allocation algorithm to cater for processes with precedence relationships. This extension could also possibly cater for heterogeneous network topologies.

The static allocation scheme presented may not be enough for load balancing. It may therefore be necessary to implement a dynamic allocation scheme. This scheme will then cater for precedence relationships between modules of a program. This scheme will monitor processes, the availability of tuples in each tuple group and processor idle times. Potential bottleneck processes with readily available "input" tuples can then be replicated onto less active processors.

Another interesting area of parallel processing currently being investigated, is dynamically configuring a network. With the *Maxi-cluster* MC^2 transputer network recently acquired at Rhodes University, the static allocation algorithm can be modified such that, it suggests the topology. This modification may also compute the required replications of each component.

Chapter 11
Conclusion

Although Linda is inherently a shared memory model, this thesis has addressed ways of implementing it efficiently on distributed memory architecture. The tuple space analysis module developed makes it possible to distribute tuple space over selected nodes in a network. Two main criticisms of Linda, namely massive run-time overheads and massive communication overheads have been redressed by this work.

The first part has dealt with reducing the run-time overheads of tuple retrieval for both centralized and distributed tuple space systems. This has been done by partitioning tuple space into mutually exclusive partitions by using tuple-template matching as the criterion. Two algorithms were developed. The latter one has been chosen. It has a firm graph theoretic foundation and also caters for some tricky cases that the former did not handle. A matching analysis further reduces the matching overheads, eliminating run-time matching in a number of cases. This has been done by providing appropriate storage schemes for the tuple partitions. The partitioning also makes it possible to prevent processes accessing data groups they do not use.

Redundant fields within tuple partitions are also removed to reduce the length of messages transferred between components. The module also reports some programming errors and provides information on components which communicate with one another to be used in determining the communication costs needed for the second half.

The second half which is only possible after partitioning, has addressed the problem of massive communication overhead. A fast heuristic algorithm has been developed to allocate processes and tuple partitions to processors at compile-time. This algorithm minimises the communication between processors while balancing the processor loads. The algorithm assumes that apart from processors controlling peripheral hardware, the rest of the network is homogeneous. The algorithm works for any arbitrary fixed topology.

The algorithm placement has been shown to be comparable with hand placements for result, agenda and specialist parallelism Linda algorithms. It has also been shown that the algorithm is likely to outperform a manual placement for complicated programs.

The main weakness of the algorithm is the assumption of smooth continuous processing loads. But, from the way Linda programs are usually written, a run-time deviation from this is a weakness of the system, which can be rectified by the programmer seeking a more efficient solution [Car89b]. As a means to rectify this, two possible extensions to this thesis are discussed in chapter 10. These are the cost pre-processor or the dynamic allocation scheme [Han91] for which this work forms the basis. The development of the cost pre-processor is needed to fully automate the allocation of components. This will simplify the programmer's task

and fully hide the underlying Rhoda implementation from the programmer.

Bibliography

[Ada83] Ada Reference Manual, Ichbiah J. (1983), *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD-1815A.

[Ada89] Adam R. Nobil, Wontmann C. John. (1989), *Security-Control Methods for Statistical Database.* ACM Computing surveys. Vol 21, No.4, Dec 1989. pp 515-556.

[Ahu86] Ahuja S., Carriero N., Gerlenter D. (1986) *Linda and Friends.* Computer, IEEE Computer Society Aug '86 pp 26 - 34.

[Ahu88] Ahuja S., Carriero N., Gerlenter D., Krishnaswamy V. (1988) *Matching Language and Hardware for Parallel Computation in the Linda Machine.* IEEE Trans Computers, 37(8), Aug 1988 pp 921-929.

[And82] Andrews G.R, (1982) *The Distributed Programming Language SR - Mechanisms, Design and Implementation.* Software - Practice and Experience, 12(8), pp 719-754.

[Bal89] Bal E. Henri, Steiner G. Jennifer, Tanenbaum S. Andrew. (1989) *Programming Languages for Distributed Computing Systems.* Vol 21, No.3 Sept 1989. pp 261-322.

[Ber73] Berztiss A.T, (1973) *Data Structures Theory and Practice.* Academic Press New York and London.

[Bjo87] Bjornson R., Carriero N. J., Gerlenter D. H., and Leichter J. (1987) *Linda, The Portable Parallel.* Yale Univ. Dept. of Computer Science Research Report 520, Feb. 1987.

[Bri75] Brinch Hansen P, (1975) *The Programming Language Concurrent Pascal.* IEEE Trans. Software Engineering 1(2) pp 199 - 206.

[Car85] Carriero Nicholas and Gerlenter David. (1985) *The S/Net's Linda Kernel.*

[Car86] Carriero Nicholas and Gerlenter David, (1986) *Distributed Data Structures in Linda.* 1986 ACM-0-89791-175-X-1/86-0236, pp 236 - 242.

[Car87] Carriero Nicholas, (1987) *Implementing Tuple Space machines.* Yale Univ. Dept of Computer Science Research Report 567, Dec. 1987.

[Car88] Carriero Nicholas and Gerlenter David, (1988 *Applications Experience with Linda.* 1988 ACM 0-89791-276-4/0007/0173. pp 173 - 187.

[Car89a] Carriero Nicholas and Gerlenter David, (1989) *Linda in Context.* Communications of the ACM, April 89 vol 32 Number 4, pp 444 - 458.

[Car89b] Carriero Nicholas and Gerlenter David, (1989) *How to Write Parallel Programs:A Guide to the Perplexed.* ACM Computing Surveys, Vol 21, No.3, September 1989. pp 323 - 356.

[Chu80] Chu W.W., Holloway L.J., Lan M., Efe K. (1980) *Task Allocation in Distributed Data Processing.* Computer Vol 13, No.11, pp 57-69.

[Chu87a] Chu W.W., Leung K.K. (1987) *Module Replication and Assignment for Real-Time Distributed Processing Systems.* Proceedings of the IEEE. Vol 75, No. 5, May 1987.

[Chu87b] Chu W. W., Sit C-M. (1987) *A Batch Service Scheduling Algorithm with Time-Out for Real-*

Time Distributed Processing Systems. Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, West Germany, September 21-25, 1987, 250-257.

[Cla89] Clayton P. G., (1989) *An Overview of parallel programming paradigms for sharing information in distributed systems.* Department of Computer Science, Rhodes University.

[Cla90] Clayton P.G., Wentworth E.P., Wells G.C., de-Heer-Menlah F.K., (1990) *An Implementation of Linda Tuple Space under the Helios Operating System.* Tech Doc 90/8. Dept of Computer Science. Rhodes University.

[Coo80] Cook R.P, (1980) **MOD - A Language for Distributed Programming.* IEEE Trans. Software Eng., 6(6), pp 563 - 571.

[Cri86] Cripps M. D, Field A. J, Reeve M.J. (1986) *The Design and Implementation of Alice: A Parallel Graph Reduction Machine.* Proc. Working Conference BCS Parallel Processing Specialist Group: Languages for Parallel Processing, Imperial College London, Sept.

[Dan90] Mathew Daniel, (1990) *A Linda Device on a Local Area Network.* Honours project Computer Science Dept, Rhodes University 1990.

[Dav89] Davidson Craig, (1989) *Technical Correspondence.* Communications of the ACM Oct 1989. Vol 32. No.10, pp 1249 - 1251.

[deh90a] de-Heer-Menlah F.K, (1990) *An investigation into the rotation of 3-D objects on the IBM-AT and a parallel transputer network.* Technical Document 90/2, Computer Science Dept. Rhodes University.

[deh90b] de-Heer-Menlah F.K, (1990) *Analyzing communication flow and process placement in Linda programs on Transputers.* Proceedings of fifth National Masters and Phd Computer Science Students Conference. RSA. 30 August - 2 September 1990. pp 39 - 45.

[deh90c] de-Heer-Menlah F.K, (1990) *Analyzing communication flow and process placement in Linda programs on Transputers.* Technical Document 90/4, Computer Science Dept Rhodes University.

[Deo74] Deo Narsingh, (1974) *Graph Theory With Applications to Engineering and Computer Science.* Prentice-Hall.

[Dik89] Dikshit P, Tripathi S.K. and Jalote P., (1989) *SAHAYOG: A Test Bed for Evaluating Dynamic Load-sharing Policies.* Software-Practice and Experience, vol. 19(5). pp 411-435 May 1989.

[Dur89] Durham Tony, (1989) *Communicating Linda's message to the world.* Computing, October 1989. pp 32 & 33.

[Efe82] Efe K. (1982) *Heuristic Models of Task Assignment Scheduling in Distributed Systems.* Computer IEEE Computer Society, Vol.15, 50-56.

[Eve80] Everitt Brian, (1980) *Clustering Analysis.* John Wiley & Sons, New York.

[Elm90] Elmagarmid K. Ahmed, Pu Calton. (1990) *Introduction to Special Issue on Heterogeneous Databases.* ACM Computing Surveys, Vol 22, No.3, pp 175-178.

[Faa90] Faasen C.R. (1990) ***Linda - An Overview and Proposed Transputer-Based Implementation.*** Proceedings of fifth National Masters and Phd Computer Science Students Conference. RSA. 30 August - 2 September 1990. pp 100 - 116.

[Fin88] Finkel A. Rapheal. (1988) ***Large-Grain Parallelism – Three case studies.*** Scientific Computation Series. MIT press. ISBN 0-262-10036-3 The Characteristics of Parallel Algorithms. pp 21-64.

[Fly66] Flynn M.J. (1966) ***Very high-speed computing systems.*** Proceedings of the IEEE 54, 12 (December) pp 1901 - 1909.

[Gaj85] Gajski D.D, Peir J. (1985) ***Essential Issues in Multiprocessor systems.*** Computer, vol. 18, No 6. pp 9 - 27.

[Gel85] Gerlenter David, (1985) ***Generative Communication in Linda.*** ACM Transactions on Programming Languages and Systems, vol 7, No. 1, January 1985, pp 80 - 112.

[Gel88] Gerlenter David, (1988) ***Getting the Job Done.*** BYTE November 1988 pp 301 - 308.

[Gel89] Gerlenter David, (1989) ***The Metamorphosis of Information Management.*** Scientific American, August 1989. pp 54 -61.

[Gri76] Gries David, (1976) ***Compiler Construction for digital Computers.*** John Wiley & Sons, Inc. pp 36-39.

[Gyl76] Gylys V.B., Edwards J.A. (1976) ***Optimal Partitioning of Workload for Distributed Systems.*** Digest of Papers, IEEE Computer Society COMPCON '76 Proceedings, pp 353-357.

[Han89] Handler C. (1989) ***Parallel Process Placement.*** MSc. (Applied Computer Science) thesis, Department of Computer Science, Rhodes University, Grahamstown.

[Han91] Handler C, de-Heer-Menlah F.K, Wentworth E.P. (1991) ***Load Balancing - A Linda Approach.*** Proceedings of IEEE Symposium on Parallel Processing, Design and Implementation CSIR Conference centre Pretoria June 1991.

[Hel89] Perihelion Software (1989) ***The Helios Operating System***. Prentice Hall.

[Hil88] Hill D.T.(1986) ***Towards a Portable Occam.*** Msc. (Applied Computer Science) thesis, Department of Computer Science, Rhodes University, Grahamstown.

[Hoa78] Haore C.A.R. (1978) ***Communicating Sequential Processes.*** Comm. ACM, 21(8) pp 666 - 667.

[Hor77] Horst A.Eiselt, Helmut von Frajer, (1977) ***Operations Research Handbook. Standard Algorithms and Methods.*** Walter de Gruyter.Berlin.New York.

[Ian89] Iannello G, Mazzeo A, Ventre G. (1989), ***Definition of the DISC Concurrent Language.*** SIGPLAN Notices, 24(6), pp 59 - 68.

[Iee88] IEEE. (1988) ***The Proposed POSIX Standard.*** IEEE Std 1003.1

[Inm88] INMOS Ltd. (1988) ***Occam 2 Reference Manual.*** Prentice-Hall, Englewood Cliffs,NJ.

[Jam88] Jamieson H. Leah (1988) ***Characterizing Parallel Algorithms.*** Scientific Computation Series. MIT press. The Characteristics of Parallel Algorithms. ISBN 0-262-10036-3. pp 65-100.

[Jar84] Jarke Matthias, Koch Jurgen. (1984) *Query Optimization in Database Systems.* ACM Computing Surveys, Vol 16, No.2, June 1984. pp 111-152.

[Kas84] Kasahara H, Narita S. (1984) *Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing.* IEEE Transactions on Computers, Vol c-33, No.11 pp 1023-1029.

[Kim84] Kim Won. (1984) *Highly Available systems for database Applications.* ACM Computing Surveys Vol 16, No.1, March 1984, pp 71-98.

[Kin83] King L. John. (1983) *Centralized versus Decentralized Computing: Organizational Considerations and Management options.* ACM computing surveys, Vol 15, No.4 Dec 1983. pp 319-350.

[Lel90] Leler W.M, (1990) *Linda meets Unix.* Computer, 23(4), pp 43 - 45.

[Lit90] Litwig Witold, Mark Leo, Roussopoulos Nick. (1990) *Interoperability of Multiple Autonomous Databases.* ACM Computing Surveys, Vol 22, No.3, pp 267-293.

[Mac89] Macharia G.M., (1989) *A Novel Approach to Dynamic Load Balancing.* North-Holland Microprocessing and Microprogramming 28 (1989) pp 43-48.

[Mao80] Mao T.W, Yeh R.T, (1980) *Communication Ports: A Language Concept for Concurrent Programming.* IEEE Trans. Software Eng. 6(2) pp 194 - 204.

[Map82] Ma, P.R. (1982) *A task Allocation Model for Distributed Computing Systems.* IEEE Transactions on Computers. vol c-31, No.1, pp 41-47.

[Mat88] Matsuoka S., Kawai S., (1988) *Using Tuple Space Communication in Distributed Object-Oriented Languages.* 1988 ACM 0-89791-284-5/88/00090276. pp 276 - 284.

[McK88] McKeown G.P. and Rayward-Smith V.J., (1988) *On process assignment in parallel computing.* Information Processing Letters 29(1988) pp 31 - 34.

[Nan90] Nandan U, Stiles G.S, (1990) *A Simple but Flexible Model for Determining Optimal Task Allocation and Configuration on a Network of Transputers.* Research report, Dept of Electrical Engineering, Utah State University.

[Nel88] Nelson P.A, Snyder L, (1988) *Programming Paradigms for Nonshared Memory Parallel Computers.* Scientific Computation Series. MIT press. The Characteristics of Parallel Algorithms. ISBN 0-262-10036-3. pp 3-20.

[Qui84] Quinn M.J. and Deo N., (1984) *Parallel Graph Algorithms.* Computing Surveys, vol. 16, No. 3, Sept 1984.

[Qui88] Quinn M.J. (1988) *Designing efficient algorithms for Parallel computers.* McGraw-Hill Book Company.

[Sad90] Sadayappan P, Ercal F, Ramanujam J. (1990) *Cluster partitioning approaches to mapping parallel programs onto a hypercube.* Parallel Computing 13 (1990) North-Holland. pp 1 - 16.

[Sch89] Schwan K, Blake B, Bo W, Gawkowski J, (1989) *Global data and control in multicomputers: operating system primitives and experimentation with a parallel branch-and-bound algorithm.* Concurrency: Practice and Experience, vol 12 Dec 1989. pp 191 - 218.

[She90a] Shekhar K. H, Srikant Y. N, (1990) *Linda Subsystem on Transputers.*

[She90b] Sheth P. Amit, Larson A. James, (1990) *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases.* ACM Computing Surveys, Vol 22, No.3, September 1990. pp 183-236.

[Shi89] Shirazi B., Wang M. (1989) *Heuristic Functions for Static Task Allocation.* North-Holland Microprocessing and Microprogramming, 26(1989) pp 187 - 194.

[Siv89] Siva C., Rajaraman V. (1989) *Task Assignment in a Multi-processor System.* North-Holland Microprocessing and Microprogramming, 26(1989) pp 63 - 71.

[Tan81] Tanenbaum A.S. (1981) *Computer Networks.* Prentice-Hall Inc, Englewood Cliffs.

[Tho90] Thomas G, Thompson G. R, Chung C.-W, Barkmeyer E, Carter F, Templeton, Fox S, Hartman B, (1990) *Heterogeneous Distributed Database Systems for Production Use.* ACM Computing Surveys, vol 22, No.3 pp 237-266.

[Tra89] Inmos Limited (1989) *The Transputer DataBook.* Prentice Hall.

[Tyr89] Tyrrell A.M. and Nicoud J.D.(1989) *Scheduling and Parallel Operations on the Transputer.* North-Holland Microprocessing and Microprogramming 26 (1989) pp 175 - 185.

[Vra89] Vranes S. (1989) *A heuristic Algorithm for Real-Time Application Allocation to Multi Microcomputer.* North-Holland. Microprocessing and Microprogramming 25 (1989) 347 - 352.

[Wan90] Wang C.M, and Wang S.D. (1990) *Structured partitioning of concurrent programs for execution on multiprocessors.* Parallel Computing. 16(1990) pp 41 - 57. North-Holland.

[War85] Ward P.T, and Mellor S.J. (1985) *Structured Development for Real-Time System.* vols. 1-3. Yourdon Press.

[Wel90] Wells G. C, (1990) *An Implementation of Linda.* Proceedings of fifth National Masters and Phd Computer Science Students Conference. RSA. 30 August - 2 September 1990. pp 302 - 307.

[Wen89] Wentworth E.P., (1989) *Prototyping a Linda System.* Status Report, Department of Computer Science. Rhodes University September 1989.

[Wir77] Wirth N. (1977) *Modula: A language for Modular Multiprogramming.* Software - Practice and Experience. 7(1) pp 3 - 35.

[Zen90] Zenith S. E., (1990) *Linda coordination language; subsystem kernel architecture (on transputers).* Research Report YALEU/DCS/RR-794 May 1990.

Appendix 1
Helios

Helios is a UNIX-like operating system for multi-processor machines. The Helios nucleus, which must be present on all Helios processors, provide a small kernel for managing message passing, hardware resources, and list handling as well as a number of basic servers which integrate the processors into the global environment. The Helios operating system includes a UNIX-compatible library, which is based on the POSIX [Iee88] standard.

The kernel provided by Helios includes a number of basic servers or utilities which integrate a processor into the global Helios environment [Hel89]. The minimum set of servers required by a Helios processor includes a loader, a processor manager, and a number of **I/O controller** (IOC) processes. The processor manager manages the computing resources of the processor and responds to requests to execute tasks on the processor. Specific operating system servers might be loaded on individual processors of the network to support certain facilities. These include the network server responsible for distributing and controlling the nucleus, a window server, the disk server, the RS232 server, the console server, and others. Most importantly, Helios provides a server library facility which can be used to implement additional servers for the system using the standard GSP protocols.

The Helios server library provides support for a message decoder and dispatcher. The server essentially consists of the dispatcher process which waits for messages to arrive on the server's request port. To handle a request, the dispatcher spawns a separate process to execute the required function. Normally, this process returns a reply at the end of the desired operation and terminates. However, if the spawned service process is an *open* operation, the service process remains active after a reply has been sent, and acts as a proxy server for any stream of messages which are directed to it, until it is closed.

Helios uses an hierarchical naming scheme for all objects in a network. Each cluster is given a unique sub-network name. The names of network objects (processors, files, file systems, servers tasks and so on) within clusters must not conflict when they are identified by their position in the network hierarchy. All objects on a processor present a directory interface through which any information specific to them may be examined and manipulated.

Most Helios commands which access the hierarchical directory structure are generic utilities which do not differentiate between the different types of objects in the hierarchy. Helios servers are written as a set of calls to a distributed server library, plus a set of application-specific functions which all adhere to the GSP protocols.